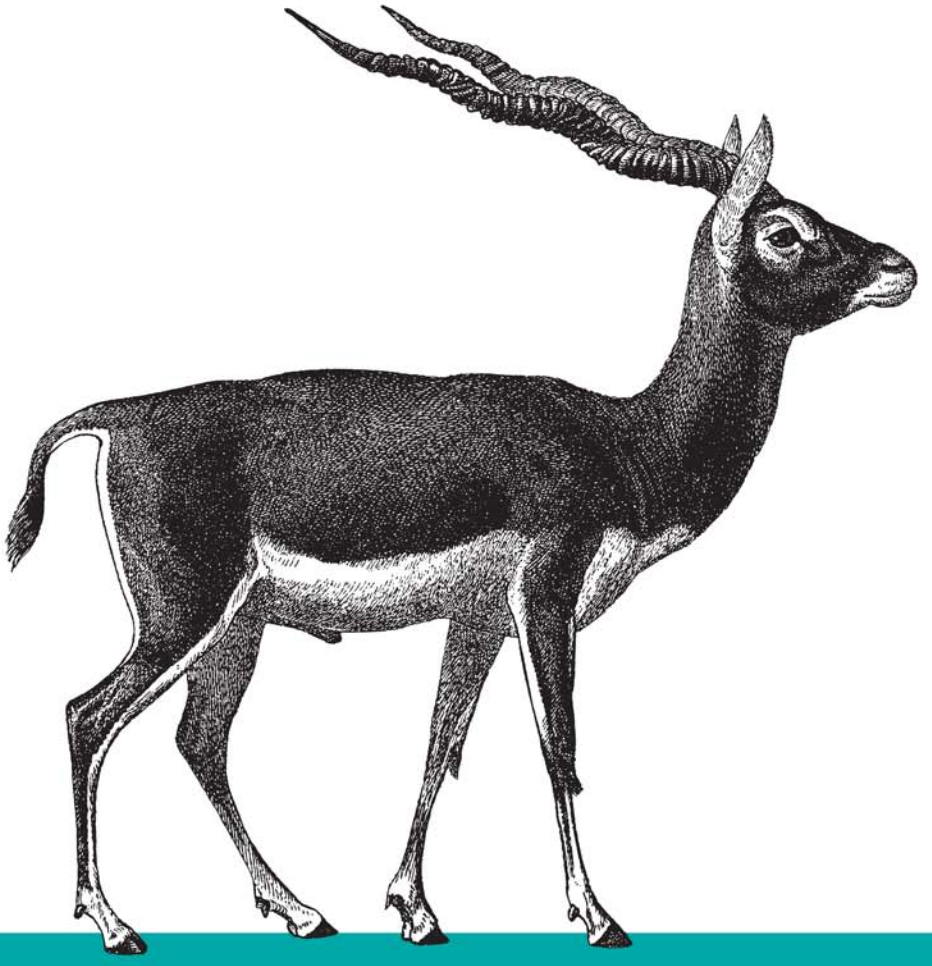
Performance Best Practices for Web Developers



Even Faster Web Sites

O'REILLY®

Steve Souders

O'REILLY®

Even Faster Web Sites

Performance is critical to the success of any web site, and yet today's web applications push browsers to their limits with increasing amounts of rich content and heavy use of Ajax. In this book, Steve Souders, web performance evangelist at Google and former Chief Performance Yahoo!, provides valuable techniques to help you optimize your site's performance.

Souders' previous book, the bestselling *High Performance Web Sites*, shocked the web development world by revealing that 80% of the time it takes for a web page to load is on the client side. In *Even Faster Web Sites*, Souders and eight expert contributors provide best practices and pragmatic advice for improving your site's performance in three critical categories:

- **JavaScript—Get advice for understanding Ajax performance, writing efficient JavaScript, creating responsive applications, loading scripts without blocking other components, and more.**
- **Network—Learn to share resources across multiple domains, reduce image size without loss of quality, and use chunked encoding to render pages faster.**
- **Browser—Discover alternatives to iframes, how to simplify CSS selectors, and other techniques.**

Speed is essential for today's rich media web sites and Web 2.0 applications. With this book, you'll learn how to shave precious seconds off your sites' load times and make them respond even faster.

"Even Faster Web Sites *has the latest up-to-date, curated wisdom on how to make your site scream. I loved the book's format—lots of topics, each explored deeply by some of the most respected authorities in the field. Everyone on my team will own a copy.*"

—**Bill Scott, Director, UI Engineering, Netflix**

Steve Souders works at Google on web performance and open source initiatives. He is the creator of YSlow, the performance analysis extension to Firebug, and is cochair of Velocity, O'Reilly's web performance and operations conference. Steve frequently speaks at conferences and at high-level companies including Microsoft, Amazon, MySpace, LinkedIn, and Facebook.

Contributing authors:
Dion Almaer, Douglas Crockford, Ben Galbraith, Tony Gentilcore, Dylan Schiemann, Stoyan Stefanov, Nicole Sullivan, and Nicholas C. Zakas

oreilly.com

US $34.99 **CAN $43.99**
ISBN: 978-0-596-52230-8

Safari Books Online

Free online edition for 45 days with purchase of this book. Details on last page.

Even Faster Web Sites

Even Faster Web Sites

Steve Souders

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Even Faster Web Sites
by Steve Souders

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mary E. Treseler
Production Editor: Sarah Schneider
Copyeditor: Audrey Doyle
Proofreader: Sarah Schneider

Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
June 2009: First Edition.

ISBN: 978-0-596-52230-8

[M]

1243719104

Table of Contents

Credits

Even Faster Web Sites contains six chapters contributed by the following authors.

Dion Almaer is the cofounder of Ajaxian.com, the leading source of the Ajax community. For his day job, Dion coleads a new group at Mozilla focusing on developer tools for the Web, something he has been passionate about doing for years. He is excited for the opportunity, and he gets to work with Ben Galbraith, his partner in crime on Ajaxian and now at Mozilla. Dion has been writing web applications since Gopher, has been fortunate enough to speak around the world, has published many articles and a book, and, of course, covers life, the universe, and everything else on his blog at *http://almaer.com/blog*.

Douglas Crockford was born in the wilds of Minnesota, but left when he was only six months old because it was just too damn cold. He turned his back on a promising career in television when he discovered computers. He has worked in learning systems, small business systems, office automation, games, interactive music, multimedia, location-based entertainment, social systems, and programming languages. He is the inventor of Tilton, the ugliest programming language that was not specifically designed to be an ugly programming language. He is best known for having discovered that there are good parts in JavaScript. This was an important and unexpected discovery. He discovered the JSON (JavaScript Object Notation) data interchange format. He is currently working on making the Web a secure and reliable software-delivery platform. He has his work cut out for him.

Ben Galbraith is the codirector of developer tools at Mozilla and the cofounder of Ajaxian.com. Ben has long juggled interests in both business and tech, having written his first computer program at 6 years old, started his first business at 10, and entered the IT workforce at 12. He has delivered hundreds of technical presentations worldwide, produced several technical conferences, and coauthored more than a half-dozen books. He has enjoyed a variety of business and technical roles throughout his career, including CEO, CIO, CTO, and Chief Software Architect roles in medical, publishing, media, manufacturing, advertising, and software industries. He lives in Palo Alto, California with his wife and five children.

Tony Gentilcore is a software engineer at Google. There, he has helped make the Google home and search results pages lightning fast. He finds that the days seem to fly by while writing web performance tools and techniques. Tony is also the creator of the popular Firefox extension, Fasterfox.

Dylan Schiemann is CEO of SitePen and cofounder of the Dojo Toolkit, an open source JavaScript toolkit for rapidly building web sites and applications, and is an expert in the technologies and opportunities of the Open Web. Under his guidance, SitePen has grown from a small development firm to a leading provider of inventive tools, skilled software engineers, knowledgeable consulting services, and top-notch training and advice. Dylan's commitment to R&D has enabled SitePen to be a major contributor to and creator of pioneering open source web development toolkits and frameworks such as Dojo, cometD, Direct Web Remoting (DWR), and Persevere. Prior to SitePen, Dylan developed web applications for companies such as Renkoo, Informatica, Security FrameWorks, and Vizional Technologies. He is a cofounder of Comet Daily, LLC, a board member at Dojo Foundation, and a member of the advisory board at Aptana. Dylan earned his master's in physical chemistry from UCLA and his B.A. in mathematics from Whittier College.

Stoyan Stefanov is a Yahoo! frontend developer, focusing on web application performance. He is also the architect of the performance extension YSlow 2.0 and cocreator of the Smush.it image optimization tool. Stoyan is a speaker, book author (*Object-Oriented JavaScript* from Packt Publishing), and blogger at *http://phpied.com*, *http://jspatterns.com*, and YUIblog.

Nicole Sullivan is an evangelist, frontend performance consultant, and CSS Ninja. She started the Object-Oriented CSS open source project, which answers the question, How do you scale CSS for millions of visitors or thousands of pages? She also consulted with the W3C for their beta redesign, and she is the cocreator of Smush.it, an image optimization service in the cloud. She is passionate about CSS, web standards, and scalable frontend architecture for large commercial websites. Nicole speaks about performance at conferences around the world, most recently at The Ajax Experience, ParisWeb, and Web Directions North. She blogs at *http://stubbornella.org*.

Nicholas C. Zakas is the author of *Professional JavaScript for Web Developers*, Second Edition (Wrox) and coauthor of *Professional Ajax*, Second Edition (Wrox). Nicholas is principal frontend engineer for the Yahoo! home page and is also a contributor to the Yahoo! User Interface (YUI) library. He blogs regularly at his site, *http://www.nczonline.net*.

Preface

Vigilant: alertly watchful, especially to avoid danger

Anyone browsing this book—or its predecessor, *High Performance Web Sites*—understands the dangers of a slow web site: frustrated users, negative brand perception, increased operating expenses, and loss of revenue. We have to constantly work to make our web sites faster. As we make progress, we also lose ground. We have to be alert for the impact of each bug fix, new feature, and system upgrade on our web site's speed. We have to be watchful, or the performance improvements made today can easily be lost tomorrow. We have to be vigilant.

Vigil: watch kept on a festival eve

According to the Latin root of *vigil*, our watch ends with celebration. Web sites can indeed be faster—dramatically so—and we can celebrate the outcome of our care and attention. It's true! Making web sites faster is attainable. Some of the world's most popular web sites have reduced their load times by 60% using the techniques described in this book. Smaller web properties benefit as well. Ultimately, users benefit.

Vigilante: a self-appointed doer of justice

It's up to us as developers to guard our users' interests. At your site, evangelize performance. Implement these techniques. Share this book with a coworker. Fight for a faster user experience. If your company doesn't have someone focused on performance, appoint yourself to that role. *Performance vigilante*—I like the sound of that.

How This Book Is Organized

This book is a follow-up to my first book, *High Performance Web Sites* (O'Reilly). In that book, I lay out 14 rules for better web performance:

- Rule 1: Make Fewer HTTP Requests
- Rule 2: Use a Content Delivery Network
- Rule 3: Add an `Expires` Header
- Rule 4: Gzip Components

- Rule 5: Put Stylesheets at the Top
- Rule 6: Put Scripts at the Bottom
- Rule 7: Avoid CSS Expressions
- Rule 8: Make JavaScript and CSS External
- Rule 9: Reduce DNS Lookups
- Rule 10: Minify JavaScript
- Rule 11: Avoid Redirects
- Rule 12: Remove Duplicate Scripts
- Rule 13: Configure ETags
- Rule 14: Make Ajax Cacheable

I call them "rules" because there is little ambiguity about their adoption. Consider these statistics for the top 10 U.S. web sites* for March 2007:

- Two sites used CSS sprites.
- 26% of resources had a future `Expires` header.
- Five sites compressed their HTML, JavaScript, and CSS.
- Four sites minified their JavaScript.

The same statistics for April 2009 show that these rules are gaining traction:

- Nine sites use CSS sprites.
- 93% of resources have a future `Expires` header.
- Ten sites compress their HTML, JavaScript, and CSS.
- Nine sites minify their JavaScript.

The rules from *High Performance Web Sites* still apply and are where most web companies should start. Progress is being made, but there's still more work to be done on this initial set of rules.

But the Web isn't standing still, waiting for us to catch up. Although the 14 rules from *High Performance Web Sites* still apply, the growth in web page content and Web 2.0 applications introduces a new set of performance challenges. *Even Faster Web Sites* provides the best practices needed by developers to make these next-generation web sites faster.

The chapters in this book are organized into three areas: JavaScript performance (Chapters 1–7), network performance (Chapters 8–12), and browser performance (Chapters 13 and 14). A roundup of the best tools for analyzing performance comes in the Appendix.

* AOL, eBay, Facebook, Google Search, Live Search, MSN.com, MySpace, Wikipedia, Yahoo!, and YouTube, according to Alexa.

Six of the chapters were written by contributing authors:

- Chapter 1, *Understanding Ajax Performance*, by Douglas Crockford
- Chapter 2, *Creating Responsive Web Applications*, by Ben Galbraith and Dion Almaer
- Chapter 7, *Writing Efficient JavaScript*, by Nicholas C. Zakas
- Chapter 8, *Scaling with Comet*, by Dylan Schiemann
- Chapter 9, *Going Beyond Gzipping*, by Tony Gentilcore
- Chapter 10, *Optimizing Images*, by Stoyan Stefanov and Nicole Sullivan

These authors are experts in each of these areas. I wanted you to hear from them directly, in their own voices. To help identify these chapters, the name(s) of the contributing author(s) are on the chapter's opening page.

JavaScript Performance

In my work analyzing today's web sites, I consistently see that JavaScript is the key to better-performing web applications, so I've started the book with these chapters.

Douglas Crockford wrote Chapter 1, *Understanding Ajax Performance*. Doug describes how Ajax changes the way browsers and servers interact, and how web developers need to understand this new relationship to properly identify opportunities for improving performance.

Chapter 2, *Creating Responsive Web Applications*, by Ben Galbraith and Dion Almaer, ties JavaScript performance back to what really matters: the user experience. Today's web applications invoke complex functions at the click of a button and must be evaluated on the basis of what they're forcing the browser to do. The web applications that succeed will be written by developers who understand the effects of their code on response time.

I wrote the next four chapters. They focus on the mechanics of JavaScript—the best way to package it and load it, and where to insert it in your pages. Chapter 3, *Splitting the Initial Payload*, describes the situation facing many web applications today: a huge JavaScript download at the beginning of the page that blocks rendering as well as further downloads. The key is to break apart this monolithic JavaScript for more efficient loading.

Chapters 4 and 5 go together. In today's most popular browsers, external scripts block everything else in the page. Chapter 4, *Loading Scripts Without Blocking*, explains how to avoid these pitfalls when loading external scripts. Loading scripts asynchronously presents a challenge when inlined code depends on them. Luckily, there are several techniques for coupling inlined code with the asynchronous scripts on which they depend. These techniques are presented in Chapter 5, *Coupling Asynchronous Scripts*.

Chapter 6, *Positioning Inline Scripts*, presents performance best practices that apply to inline scripts, especially the impact they have on blocking parallel downloads.

I think of Chapter 7, *Writing Efficient JavaScript*, written by Nicholas C. Zakas, as the complement to Doug's chapter (Chapter 1). Whereas Doug describes the Ajax landscape, Nicholas zooms in on several specific techniques for speeding up JavaScript.

Network Performance

Web applications aren't desktop applications—they have to be downloaded over the Internet each time they are used. The adoption of Ajax has resulted in a new style of data communication between servers and clients. Some of the biggest opportunities for growth in the web industry are in emerging markets where Internet connectivity is a challenge, to put it mildly. All of these factors highlight the need for improved network performance.

In Chapter 8, *Scaling with Comet*, Dylan Schiemann describes an architecture that goes beyond Ajax to provide high-volume, low-latency communication for real-time applications such as chat and document collaboration.

Chapter 9, *Going Beyond Gzipping*, describes how turning on compression isn't enough to guarantee optimal delivery of your web site's content. Tony Gentilcore reveals a little-known phenomenon that severely hinders the network performance of 15% of the world's Internet users.

Stoyan Stefanov and Nicole Sullivan team up to contribute Chapter 10, *Optimizing Images*. This is a thorough treatment of the topic. This chapter reviews all popular image formats, presents numerous image optimization techniques, and describes the image compression tools of choice.

The remaining chapters were written by me. Chapter 11, *Sharding Dominant Domains*, reminds us of the connection limits in the popular browsers of today, as well as the next generation of browsers. It includes techniques for successfully splitting resources across multiple domains.

Chapter 12, *Flushing the Document Early*, walks through the benefits and many gotchas of using chunked encoding to start rendering the page even before the full HTML document has arrived.

Browser Performance

Iframes are an easy and frequently used technique for embedding third-party content in a web page. But they come with a cost. Chapter 13, *Using Iframes Sparingly*, explains the downsides of iframes and offers a few alternatives.

Chapter 14, *Simplifying CSS Selectors*, presents the theories about how complex selectors can impact performance, and then does an objective analysis to pinpoint the situations that are of most concern.

The Appendix, *Performance Tools*, describes the tools that I recommend for analyzing web sites and discovering the most important performance improvements to work on.

Conventions Used in This Book

The following typographical conventions are used in this book:

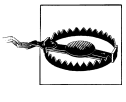
Italic
: Indicates new terms, URLs, email addresses, filenames, file extensions, pathnames, and directories

`Constant width`
: Indicates commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties, parameters, values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, and the output from commands

`Constant width bold`
: Shows commands or other text that should be typed literally by the user

`Constant width italic`
: Shows text that should be replaced with user-supplied values

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596522308

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Using Code Examples

You may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from this book *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Even Faster Web Sites*, by Steve Souders. Copyright 2009 Steve Souders, 978-0-596-52230-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

Acknowledgments

I first want to thank the contributing authors: Dion Almaer, Doug Crockford, Ben Galbraith, Tony Gentilcore, Dylan Schiemann, Stoyan Stefanov, Nicole Sullivan, and Nicholas Zakas. They've made this a special book. Each of them is an expert in his or her own right. Most of them have written their own books. By sharing their expertise, they've helped create something unique.

I want to thank all the reviewers: Julien Lecomte, Matthew Russell, Bill Scott, and Tenni Theurer. I extend an especially strong thank you to Eric Lawrence and Andy Oram. Eric reviewed both this book as well as *High Performance Web Sites*. In both cases, he provided incredibly thorough and knowledgeable feedback. Andy was my editor on *High Performance Web Sites*. More than anyone else, he is responsible for improving how this book reads, making it flow smoothly from line to line, section to section, and chapter to chapter.

A special thank you goes to my editor, Mary Treseler. Coordinating a book with multiple authors is an opportunity that many editors will pass over. I'm glad that she took on this project and helped guide it from a bunch of ideas to what you're holding in your hands now.

I work with many people at Google who have a penchant for web performance. Tony Gentilcore is the creator of Fasterfox and the author of Chapter 9. He's also my officemate. Several times a day we'll stop to discuss web performance. Steve Lamm, Lindsey Simon, and Annie Sullivan are strong advocates for performance who I work with frequently. Other Googlers who have contributed to what I know about web performance include Jacob Hoffman-Andrews, Kyle Scholz, Steve Krulewitz, Matt Gundersen, Gavin Doughtie, and Bryan McQuade.

Many of the insights in this book come from my friends outside Google. They know that if they tell me about a good performance tip, it's likely to end up in a book or blog post. These performance cohorts include Dion Almaer, Artur Bergman, Doug Crockford, Ben Galbraith, Eric Goldsmith, Jon Jenkins, Eric Lawrence, Mark Nottingham, Simon Perkins, John Resig, Alex Russell, Eric Schurman, Dylan Schiemann, Bill Scott, Jonas Sicking, Joseph Smarr, and Tenni Theurer.

I've inevitably forgotten to mention someone in these lists. I apologize, and want to thank all of you for taking the time to send me email and talk to me at conferences. Hearing your lessons learned and success stories keeps me going. It's important to know there are so many of us who are working to make the Web a faster place.

Thank you to my parents for being proud to have a son who's an author. Most importantly, thank you to my wife and three daughters. I promise to take a break now.

CHAPTER 1

Understanding Ajax Performance

Douglas Crockford

Premature optimization is the root of all evil.

—Donald Knuth

Trade-offs

The design and construction of a computer program can involve thousands of decisions, each representing a trade-off. In difficult decisions, each alternative has significant positive and negative consequences. In trading off, we hope to obtain a near optimal good while minimizing the bad. Perhaps the ultimate trade-off is:

> I want to go to heaven, but I don't want to die.

More practically, the Project Triangle:

> Fast. Good. Cheap. Pick Two.

predicts that even under ideal circumstances, it is not possible to obtain fast, good, and cheap. There must be a trade-off.

In computer programs, we see time versus memory trade-offs in the selection of algorithms. We also see expediency or time to market traded against code quality. Such trades can have a large impact on the effectiveness of incremental development.

Every time we touch the code, we are trading off the potential of improving the code against the possibility of injecting a bug. When we look at the performance of programs, we must consider all of these trade-offs.

Principles of Optimization

When looking at optimization, we want to reduce the overall cost of the program. Typically, this cost is the perceived execution time of the program, although we could

optimize on other factors. We then should focus on the parts of the program that contribute most significantly to its cost.

For example, suppose that by profiling we discover the cost of a program's four modules.

Module	A	B	C	D
Cost	54%	4%	30%	12%

If we could somehow cut the cost of Module B in half, we would reduce the total cost by only 2%. We would get a better result by cutting the cost of Module A by 10%. There is little benefit from optimizing components that do not contribute significantly to the cost.

The analysis of applications is closely related to the analysis of algorithms. When looking at execution time, the place where programs spend most of their time is in loops. The return on optimization of code that is executed only once is negligible. The benefits of optimizing inner loops can be significant.

For example, if the cost of a loop is linear with respect to the number of iterations, then we can say it is $O(n)$, and we can graph its performance as shown in Figure 1-1.

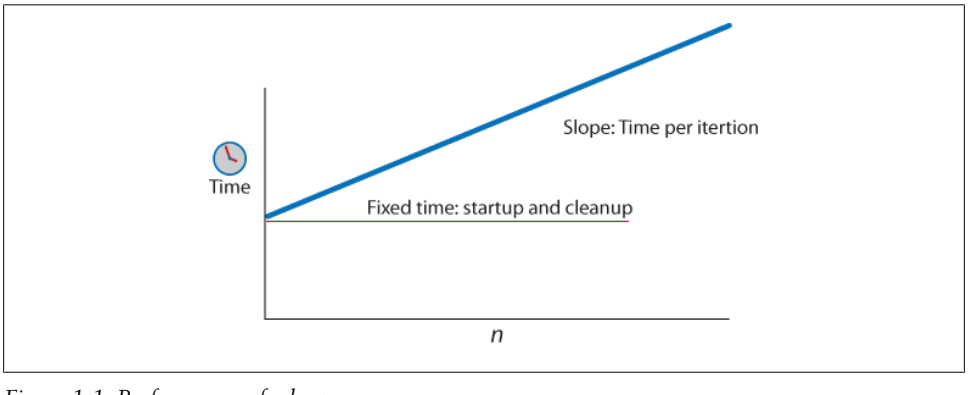


Figure 1-1. Performance of a loop

The execution time of each iteration is reflected in the slope of the line: the greater the cost, the steeper the slope. The fixed overhead of the loop determines the elevation of its starting point. There is usually little benefit in reducing the fixed overhead. Sometimes there is a benefit in increasing the fixed overhead if the cost of each increment can be reduced. That can be a good trade-off.

In addition to the plot of execution time, there are three lines—the Axes of Error—that our line must not intersect (see Figure 1-2). The first is the *Inefficiency* line. Crossing this line reduces the user's ability to concentrate. This can also make people irritable. The second is the *Frustration* line. When this line is crossed, the user is aware that he

is being forced to wait. This invites him to think about other things, such as the desirability of competing web applications. The third is the *Failure* line. This is when the user refreshes or closes the browser because the application appears to have crashed, or the browser itself produces a dialog suggesting that the application has failed and that the user should take action.

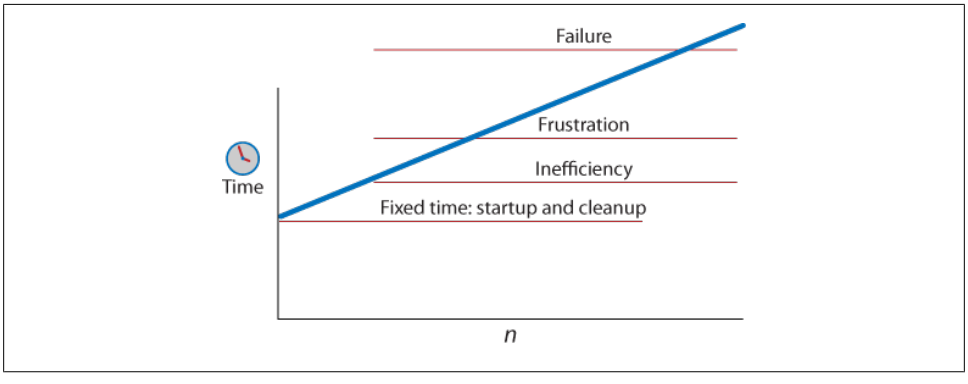


Figure 1-2. The Axes of Error

There are three ways to avoid intersecting the Axes of Error: reduce the cost of each iteration, reduce the number of iterations, or redesign the application.

When loops become nested, your options are reduced. If the cost of the loop is $O(n \log n)$, $O(n^2)$, or worse, reducing the time per iteration is not effective (see Figure 1-3). The only effective options are to reduce n or to replace the algorithm. Fiddling with the cost per iteration will be effective only when n is very small.

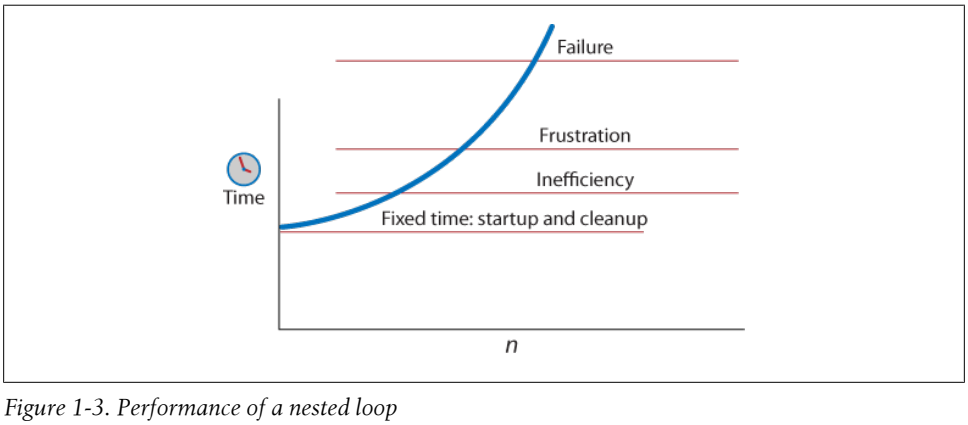


Figure 1-3. Performance of a nested loop

Programs must be designed to be correct. If the program isn't right, it doesn't matter if it is fast. However, it is important to determine whether it has performance problems as early as possible in the development cycle. In testing web applications, test with slow

machines and slow networks that more closely mimic those of real users. Testing in developer configurations is likely to mask performance problems.

Ajax

Refactoring the code can reduce its apparent complexity, making optimization and other transformations more likely to yield benefits. For example, adopting the YSlow rules can have a huge impact on the delivery time of web pages (see *http://developer.yahoo.com/yslow/*).

Even so, it is difficult for web applications to get under the *Inefficiency* line because of the size and complexity of web pages. Web pages are big, heavy, multipart things. Page replacement comes with a significant cost. For applications where the difference between successive pages is relatively small, use of Ajax techniques can produce a significant improvement.

Instead of requesting a replacement page as a result of a user action, a packet of data is sent to the server (usually encoded as JSON text) and the server responds with another packet (also typically JSON-encoded) containing data. A JavaScript program uses that data to update the browser's display. The amount of data transferred is significantly reduced, and the time between the user action and the visible feedback is also significantly reduced. The amount of work that the server must do is reduced. The amount of work that the browser must do is reduced. The amount of work that the Ajax programmer must do, unfortunately, is likely to increase. That is one of the trade-offs.

The architecture of an Ajax application is significantly different from most other sorts of applications because it is divided between two systems. Getting the division of labor right is essential if the Ajax approach is to have a positive impact on performance. The packets should be as small as possible. The application should be constructed as a conversation between the browser and the server, in which the two halves communicate in a concise, expressive, shared language. Just-in-time data delivery allows the browser side of the application to keep n small, which tends to keep the loops fast.

A common mistake in Ajax applications is to send all of the application's data to the browser. This reintroduces the latency problems that Ajax is supposed to avoid. It also enlarges the volume of data that must be handled in the browser, increasing n and again compromising performance.

Browser

Ajax applications are challenging to write because the browser was not designed to be an application platform. The scripting language and the Document Object Model (DOM) were intended to support applications composed of simple forms. Surprisingly, the browser gets enough right that it is possible to use it to deliver sophisticated

applications. Unfortunately, it didn't get everything right, so the level of difficulty can be high. This can be mitigated with the use of Ajax libraries (e.g., *http://developer.yahoo.com/yui/*). An Ajax library uses the expressive power of JavaScript to raise the DOM to a practical level, as well as repairing many of the hazards that can prevent applications from running acceptably on the many brands of browsers.

Unfortunately, the DOM API is very inefficient and mysterious. The greatest cost in running programs tends to be the DOM, not JavaScript. At the Velocity 2008 conference, the Microsoft Internet Explorer 8 team shared this performance data on how time is spent in the Alexa 100 pages.[*]

Activity	Layout	Rendering	HTML	Marshaling	DOM	Format	JScript	Other
Cost	43.16%	27.25%	2.81%	7.34%	5.05%	8.66%	3.23%	2.5%

The cost of running JavaScript is insignificant compared to the other things that the browser spends time on. The Microsoft team also gave an example of a more aggressive Ajax application, the opening of an email thread.

Activity	Layout	Rendering	HTML	Marshaling	DOM	Format	JScript	Other
Cost	9.41%	9.21%	1.57%	7.85%	12.47%	38.97%	14.43%	3.72%

The cost of the script is still less than 15%. Now CSS processing is the greatest cost. Understanding the mysteries of the DOM and working to suppress its impact is clearly a better strategy than attempting to speed up the script. If you could heroically make the script run twice as fast, it would barely be noticed.

Wow!

There is a tendency among application designers to add *wow* features to Ajax applications. These are intended to invoke a reaction such as, "Wow, I didn't know browsers could do that." When used badly, wow features can interfere with the productivity of users by distracting them or forcing them to wait for animated sequences to play out. Misused wow features can also cause unnecessary DOM manipulations, which can come with a surprisingly high cost.

Wow features should be used only when they genuinely improve the experience of the user. They should not be used to show off or to compensate for deficiencies in functionality or usability.

Design for things that the browser can do well. For example, viewing a database as an infinitely scrolling list requires that the browser hold on to and display a much larger set than it can manage efficiently. A better alternative is to have a very effective

[*] *http://en.oreilly.com/velocity2008/public/schedule/detail/3290*

paginating display with no scrolling at all. This provides better performance and can be easier to use.

JavaScript

Most JavaScript engines were optimized for quick time to market, not performance, so it is natural to assume that JavaScript is always the bottleneck. Typically, however, the bottleneck is not JavaScript, but the DOM, so fiddling with scripts will have little effectiveness.

Fiddling should be avoided. Programs should be coded for correctness and clarity. Fiddling tends to work against clarity, which can increase the susceptibility of the program to attract bugs.

Fortunately, competitive pressure is forcing the browser makers to improve the efficiency of their JavaScript engines. These improvements will enable new classes of applications in the browser.

Avoid obscure idioms that might be faster unless you can prove that they will have a noticeable impact on your application. In most cases, they will have no noticeable impact except to degrade the quality of your code. Do not tune to the quirks of particular browsers. The browsers are still in development and may ultimately favor better coding practices.

If you feel you must fiddle, measure first. Our intuitions of the true costs of a program are usually wrong. Only by measuring can you have confidence that you are having a positive effect on performance.

Summary

Everything is a trade-off. When optimizing for performance, do not waste time trying to speed up code that does not consume a significant amount of the time. Measure first. Back out of any optimization that does not provide an enjoyable benefit.

Browsers tend to spend little time running JavaScript. Most of their time is spent in the DOM. Ask your browser maker to provide better performance measurement tools.

Code for quality. Clean, legible, well-organized code is easier to get right, easier to maintain, and easier to optimize. Avoid tricks except when they can be proven to substantially improve performance.

Ajax techniques, when used well, can make applications faster. The key is in establishing a balance between the browser and the server. Ajax provides an effective alternative to page replacement, turning the browser into a powerful application platform, but your success is not guaranteed. The browser is a challenging platform and your intuitions about performance are not reliable. The chapters that follow will help you understand how to make even faster web sites.

CHAPTER 2

Creating Responsive Web Applications

Ben Galbraith and Dion Almaer

With the rise of Ajax, web site performance is no longer just about the quick realization of a web site. An ever-increasing number of web sites, once loaded, will use JavaScript to dynamically change the page and load new content on the fly. Such sites have much in common with traditional desktop client programs, and optimizing the performance of these applications requires a different set of techniques from traditional web sites.

From a high level, user interfaces for web applications and traditional desktop applications share a common goal: respond to the user's input as fast as possible. When it comes to responding to a user's request to load a web site, the browser itself handles much of the responsiveness burden. It opens network connections to the requested site, parses the HTML, requests the associated resources, and so forth. Based on a careful analysis of this process, we can optimize our pages to render as fast as possible, but the browser is ultimately in control of loading and realizing the page.

When it comes to responding to user input to the web site itself (when that input doesn't result in the browser loading a new page), we web developers are in control. We must ensure that the JavaScript that executes as a result of such input is responsive. To better understand just how much control we have over responsiveness, we're going to take a minute to explain how browser user interfaces work.

As shown in Figure 2-1, when a user interacts with a browser, the operating system receives input from various devices attached to the computer, such as the keyboard or mouse. It works out which application should receive these inputs, and it packages them up as individual events and places them in a queue for that application, known as an *event queue*.

It's up to the web browser, like any GUI application, to process the individual events placed in its queue. It does so by pulling them from the queue in first-in, first-out order and deciding what to do about the event. Generally, the browser will do one of two things based on these events: handle the event itself (such as display a menu, browse

the Web, show a preference screen, etc.) or execute JavaScript code in the web page itself (e.g., JavaScript code in an `onclick` handler in the page), as shown in Figure 2-2.

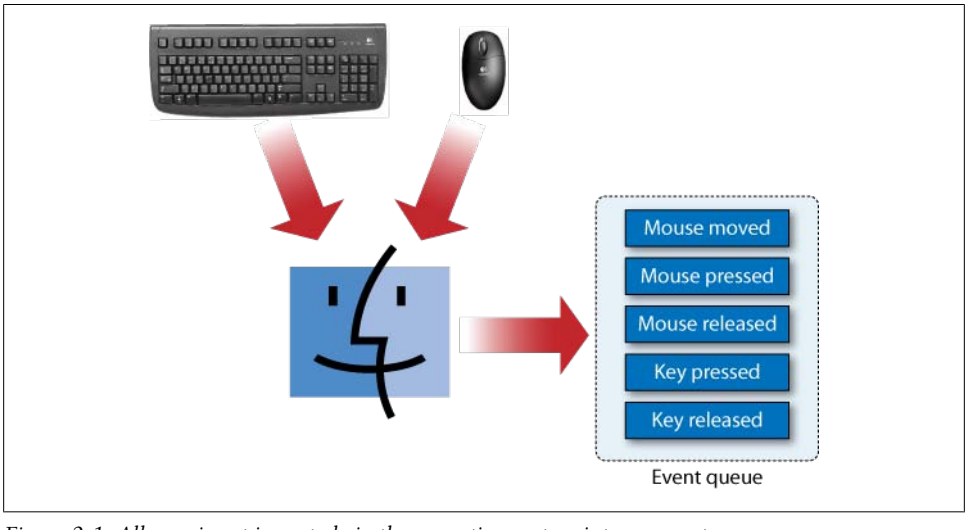


Figure 2-1. All user input is routed via the operating system into an event queue

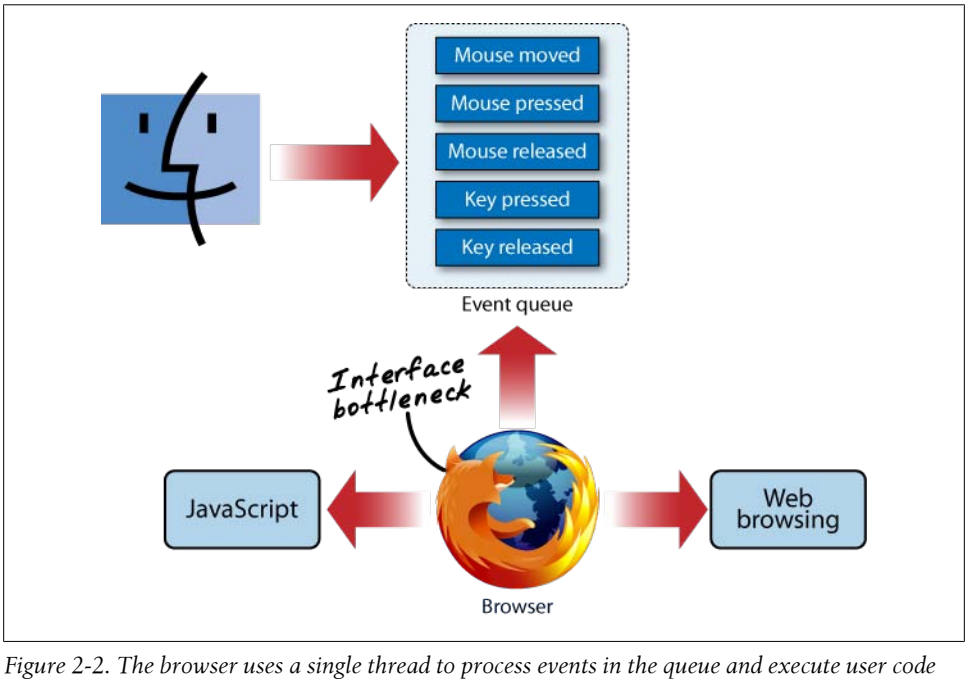


Figure 2-2. The browser uses a single thread to process events in the queue and execute user code

The important takeaway here is that this process is essentially *single-threaded*. That is, the browser uses a single thread to pull an event from the queue and either do something

itself ("Web browsing" in Figure 2-2) or execute JavaScript. As such, it can do only one of these tasks at a time, and each of these tasks can prevent the other tasks from occurring.

Any time spent by the browser executing a page's JavaScript is time that it cannot spend responding to other user events. It is therefore vital that any JavaScript in a page execute as fast as possible. Otherwise, the web page and the browser itself may become sluggish or freeze up entirely.

Note that this discussion of browser and operating system behavior with respect to input handling and events is a broadly applicable generalization; details vary. Regardless of variances, all browsers execute all JavaScript code in a page on a single thread (excepting the use of Web Workers, discussed later in this chapter), making the developer practices advocated in this chapter completely applicable.

What Is Fast Enough?

It's fine to say that code needs to execute "as fast as possible," but sometimes code needs to do things that simply take time. For instance, encryption algorithms, complex graphics rendering, and image manipulation are examples of computations that are time-consuming to perform, regardless of how much effort a developer puts forth to make them "as fast as possible."

However, as Doug mentioned in Chapter 1, developers seeking to create responsive, high-performance web sites can't—and shouldn't—go about achieving that goal by optimizing every single piece of code as they write it. The opposite is true: a developer should optimize only what isn't fast enough.

It is therefore vital to define exactly what is "fast enough" in this context. Fortunately, that's already been done for us.

Jakob Nielsen is a well-known and well-regarded expert in the field of web usability; the following quote[*] addresses the issue of "fast enough":

> The response time guidelines for web-based applications are the same as for all other applications. These guidelines have been the same for 37 years now, so they are also not likely to change with whatever implementation technology comes next.
>
> **0.1 second**: Limit for users feeling that they are directly manipulating objects in the UI. For example, this is the limit from the time the user selects a column in a table until that column should highlight or otherwise give feedback that it's selected. Ideally, this would also be the response time for sorting the column—if so, users would feel that they are sorting the table.
>
> **1 second**: Limit for users feeling that they are freely navigating the command space without having to unduly wait for the computer. A delay of 0.2–1.0 seconds does mean that users notice the delay and thus feel the computer is "working" on the command, as

[*] *http://www.useit.com/papers/responsetime.html*

opposed to having the command be a direct effect of the users' actions. Example: If sorting a table according to the selected column can't be done in 0.1 seconds, it certainly has to be done in 1 second, or users will feel that the UI is sluggish and will lose the sense of "flow" in performing their task. For delays of more than 1 second, indicate to the user that the computer is working on the problem, for example by changing the shape of the cursor.

10 seconds: Limit for users keeping their attention on the task. Anything slower than 10 seconds needs a percent-done indicator as well as a clearly signposted way for the user to interrupt the operation. Assume that users will need to reorient themselves when they return to the UI after a delay of more than 10 seconds. Delays of longer than 10 seconds are only acceptable during natural breaks in the user's work, for example when switching tasks.

In other words, if your JavaScript code takes longer than 0.1 seconds to execute, your page won't have that slick, snappy feel; if it takes longer than 1 second, the application feels sluggish; longer than 10 seconds, and the user will be extremely frustrated. These are the definitive guidelines to use for defining "fast enough."

Measuring Latency

Now that you know the threshold for fast enough, the next step is to explore how you can measure the speed of JavaScript execution to determine whether it falls outside the ranges mentioned earlier (we'll leave it to you to determine just how fast you wish your page to be; we aim to keep all interface latency smaller than 0.1 seconds).

The easiest, most straightforward, and probably least precise way to measure latency is via human observation; simply use the application on your target platforms and ensure that performance is adequate. Since ensuring adequate human interface performance is only about pleasing humans, this is actually a fine way to perform such measurements (obviously, few humans will be able to quantify delays reliably in terms of precise whole or fractional second measurements; falling back to coarser categorizations such as "snappy," "sluggish," "adequate," and so on does the job).

However, if you desire more precise measurements, there are two options you can choose: manual code instrumentation (*logging*) or automated code instrumentation (*profiling*).

Manual code instrumentation is really straightforward. Let's say you have an event handler registered on your page, as in:

```
<div onclick="myJavaScriptFunction()"> ... </div>
```

A simple way to add manual instrumentation would be to locate the definition of `myJavaScriptFunction()` and add timing to the function:

```
function myJavaScriptFunction() {
    var start = new Date().getMilliseconds();

    // some expensive code is here
```

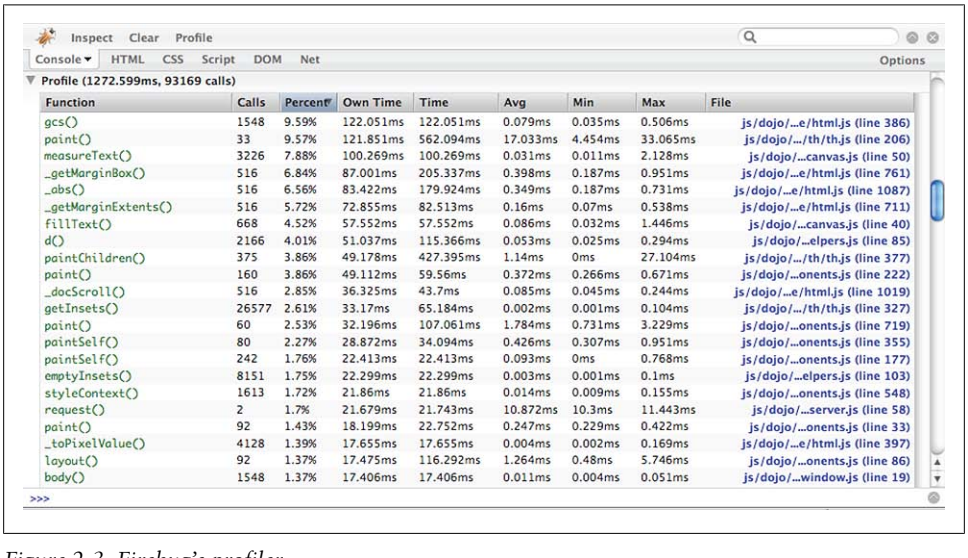
```
        var stop = new Date().getMilliseconds();
        var executionTime = stop - start;
        alert("myJavaScriptFunction() executed in " + executionTime +
            " milliseconds");
    }
```

The preceding code will produce a pop-up dialog that displays the execution time; one millisecond represents 1/1,000 of a second, so 100 milliseconds represent the 0.1-second "snappiness" threshold mentioned earlier.

Many browsers offer a built-in instance named `console` that provides a `log()` function (Firefox makes this available with the popular Firebug plug-in); we greatly prefer `console.log()` to `alert()`.

There are tools to perform an automated measurement of code execution time, but such tools are typically used for a different purpose. Instead of being used to determine the precise execution duration of a function, such tools—called *profilers*—are usually used to determine the relative amount of time spent executing a set of functions; that is, they are used to find the *bottleneck* or slowest-running chunks of code.

The popular Firebug extension for Firefox includes a JavaScript code profiler; it generates output such as that shown in Figure 2-3.



Function	Calls	Percent	Own Time	Time	Avg	Min	Max	File
gcs()	1548	9.59%	122.051ms	122.051ms	0.079ms	0.035ms	0.506ms	js/dojo/...e/html.js (line 386)
paint()	33	9.57%	121.851ms	562.094ms	17.033ms	4.454ms	33.065ms	js/dojo/.../th/th.js (line 206)
measureText()	3226	7.88%	100.269ms	100.269ms	0.031ms	0.011ms	2.128ms	js/dojo/...canvas.js (line 50)
_getMarginBox()	516	6.84%	87.001ms	205.337ms	0.398ms	0.187ms	0.951ms	js/dojo/...e/html.js (line 761)
_abs()	516	6.56%	83.422ms	179.924ms	0.349ms	0.187ms	0.731ms	js/dojo/...e/html.js (line 1087)
_getMarginExtents()	516	5.72%	72.855ms	82.513ms	0.16ms	0.07ms	0.538ms	js/dojo/...e/html.js (line 711)
fillText()	668	4.52%	57.552ms	57.552ms	0.086ms	0.032ms	1.446ms	js/dojo/...canvas.js (line 40)
d()	2166	4.01%	51.037ms	115.366ms	0.053ms	0.025ms	0.294ms	js/dojo/...elpers.js (line 85)
paintChildren()	375	3.86%	49.178ms	427.395ms	1.14ms	0ms	27.104ms	js/dojo/.../th/th.js (line 377)
paint()	160	3.86%	49.112ms	59.56ms	0.372ms	0.266ms	0.671ms	js/dojo/...onents.js (line 222)
_docScroll()	516	2.85%	36.325ms	43.7ms	0.085ms	0.045ms	0.244ms	js/dojo/...e/html.js (line 1019)
getInsets()	26577	2.61%	33.17ms	65.184ms	0.002ms	0.001ms	0.104ms	js/dojo/.../th/th.js (line 327)
paint()	60	2.53%	32.196ms	107.061ms	1.784ms	0.731ms	3.229ms	js/dojo/...onents.js (line 719)
paintSelf()	80	2.27%	28.872ms	34.094ms	0.426ms	0.307ms	0.951ms	js/dojo/...onents.js (line 355)
paintSelf()	242	1.76%	22.413ms	22.413ms	0.093ms	0ms	0.768ms	js/dojo/...onents.js (line 177)
emptyInsets()	8151	1.75%	22.299ms	22.299ms	0.003ms	0.001ms	0.1ms	js/dojo/...elpers.js (line 103)
styleContext()	1613	1.72%	21.86ms	21.86ms	0.014ms	0.009ms	0.155ms	js/dojo/...onents.js (line 548)
request()	2	1.7%	21.679ms	21.743ms	10.872ms	10.3ms	11.443ms	js/dojo/...server.js (line 58)
paint()	92	1.43%	18.199ms	22.752ms	0.247ms	0.229ms	0.422ms	js/dojo/...onents.js (line 33)
_toPixelValue()	4128	1.39%	17.655ms	17.655ms	0.004ms	0.002ms	0.169ms	js/dojo/...e/html.js (line 397)
layout()	92	1.37%	17.475ms	116.292ms	1.264ms	0.48ms	5.746ms	js/dojo/...onents.js (line 86)
body()	1548	1.37%	17.406ms	17.406ms	0.011ms	0.004ms	0.051ms	js/dojo/...window.js (line 19)

Figure 2-3. Firebug's profiler

The "Time" column represents the total amount of time the JavaScript interpreter spent inside a given function during the period of profiling. Often, a function invokes other

functions; the "Own Time" column represents the amount of time spent inside a specific function and not any other functions that it may have invoked.

While you might think these and the other temporal-related columns represent a precise measurement of function execution time, it turns out that profilers are subject to something like the *observer effect* in physics: the act of observing the performance of code modifies the performance of the code.

Profilers can take two basic strategies representing a basic trade-off: either they can intrude on the code being measured by adding special code to collect performance statistics (basically automating the creation of code as in the previous listing), or they can passively monitor the runtime by checking what piece of code is being executed at a particular moment in time. Of these two approaches, the latter does less to distort the performance of the code being profiled, but at the cost of lower-quality data.

Firebug subjects results to a further distortion because its profiler executes inside Firefox's own process, which creates the potential for it to rob the code it is measuring of performance.

Nevertheless, the "Percent" column of Firebug's output demonstrates the power of measuring relative execution time: you can perform a high-level task in your page's interface (e.g., click the Send button) and then check Firebug's profiler to see which functions spent the most time executing, and focus your optimization efforts on those.

When Latency Goes Bad

It turns out that if your JavaScript code ties up the browser thread for a particularly long time, most browsers will intervene and give the user the opportunity to interrupt your code. There is no standard behavior governing how browsers make the determination to give the user this opportunity. (For details on individual browser behavior, see *http://www.nczonline.net/blog/2009/01/05/what-determines-that-a-script-is-long-running/*.)

The lesson is simple: don't introduce potentially long-running, poorly performing code into your web page.

Threading

Once you've identified code that performs inadequately, of course the next step is to go about optimizing it. However, sometimes the task to perform is simply expensive and cannot be magically optimized to take less time. Are such scenarios fated to bring sluggish horror to a user interface? Will no solution emerge to keep our users happy?

The traditional solution in such cases is to use *threads* to push such expensive code off the thread used to interact with the user. In our scenario, this would let the browser continue to process events from the event queue and keep the interface responsive while the long-running code merrily executes on a different thread (and the operating system

takes responsibility for making sure that both the browser user interface thread and the background thread equitably share the computer's resources).

However, JavaScript doesn't support threads, so there's no way for JavaScript code to create a background thread to execute expensive code. Further, this isn't likely to change anytime soon.

Brendan Eich, the creator of JavaScript and Mozilla's chief technical officer, has made his position on this issue clear:[†]

> You must be [as tall as an NBA player] to hack on threaded systems, and that means most programmers should run away crying. But they don't. Instead, as with most other sharp tools, the temptation is to show how big one is by picking up the nearest single-threaded code and jamming it into a multi-threaded embedding, or tempting race-condition fate otherwise. Occasionally the results are infamous, but too often, with only virtual fingers and limbs lost, no one learns.
>
> Threads violate abstractions six ways to Sunday. Mainly by creating race conditions, deadlock hazards, and pessimistic locking overhead. And still they don't scale up to handle the megacore teraflop future.
>
> So my default answer to questions such as, "When will you add threads to JavaScript?" is: "over your dead body!"

Given Brendan's influence in the industry and on the future of JavaScript (which is considerable), and the broad degree to which this position is shared, it is safe to say that threads will not be coming to JavaScript anytime soon.

However, there are alternatives. The basic problem with threads is that different threads can have access to and modify the same variables. This causes all sorts of problems when Thread A modifies variables that Thread B is actively modifying, and so on. You might think these sorts of issues could be kept straight by decent programmers, but it turns out that, as Brendan said, even the best of us make pretty horrible mistakes in this department.

Ensuring Responsiveness

What's needed is a way to have the benefit of threads—tasks executing in parallel—without the hazards of the threads getting into each other's business. Google implemented just such an API in its popular Gears browser plug-in: the WorkerPool API. It essentially allows the main browser JavaScript thread to create background "workers" that receive some simple "message" (i.e., standalone state, not references to shared variables) from the browser thread when they are kicked off and return a message upon completion.

[†] *http://weblogs.mozillazine.org/roadmap/archives/2007/02/threads_suck.html*

Experience with this API in Gears has led many browsers (e.g., Safari 4, Firefox 3.1) to implement support for "workers" natively based on a common API defined in the HTML 5 specification. This feature is known as "Web Workers."

Web Workers

Let's consider how to use the Web Worker API to decrypt a value. The following listing shows how to create and kick off a worker:

```
// create and begin execution of the worker
var worker = new Worker("js/decrypt.js");

// register an event handler to be executed when the worker
// sends the main thread a message
worker.onmessage = function(e) {
    alert("The decrypted value is " + e.data);
}

// send a message to the worker, in this case the value to decrypt
worker.postMessage(getValueToDecrypt());
```

Now let's take a look at the hypothetical contents of *js/decrypt.js*:

```
// register a handler to receive messages from the main thread
onmessage = function(e) {
    // get the data passed to us
    var valueToDecrypt = e.data;

    // TODO: implement decryption here

    // return the value to the main thread
    postMessage(decryptedValue);
}
```

Any potentially expensive (i.e., long-running) JavaScript operations that your page performs should be delegated to workers, as that will keep your application running lickety-split.

Gears

If you find yourself supporting a browser that doesn't support the Web Worker API, there are a few alternatives. We mentioned Google's Gears plug-in in the preceding section; you can use the Gears plug-in to bring something very much like Web Workers to Internet Explorer, to older versions of Firefox, and to older versions of Safari.

The Gears worker API is similar but not identical to the Web Worker API. Here are the previous two code listings converted to the Gears API, starting with the code executed on the main thread to spawn a worker:

```
// create a worker pool, which spawns workers
var workerPool = google.gears.factory.create('beta.workerpool');

// register the event handler that receives the message from the worker
workerPool.onmessage = function(ignore1, ignore2, e) {
    alert("The decrypted value is + " e.body);
}

// create a worker
var workerId = workerPool.createWorkerFromUrl("js/decrypt.js");

// send a message to the worker
workerPool.sendMessage(getValueToDecrypt(), workerId);
```

And here is the Gears version of *js/decrypt.js*:

```
var workerPool = google.gears.workerPool;
workerPool.onmessage = function(ignore1, ignore2, e) {
    // get the data passed to us
    var valueToDecrypt = e.body;

    // TODO: implement decryption here

    // return the value to the main thread
    workerPool.sendMessage(decryptedValue, e.sender);
}
```

More on Gears

It is interesting to note some of the history of the Gears Worker Pool because it came from a very practical place. The Gears plug-in was built by a team at Google that was trying to push the browser to do more than it currently was able (this was before Google Chrome—but even with Chrome, Google wants as many users as possible to do great things with its web applications).

Imagine if you wanted to build Gmail Offline; what would you need? First, you'd need a way to cache documents locally and to have an intercept so that when the browser tries to access *http://mail.google.com/*, it gets the page back instead of a message stating that you are offline. Second, it needs a way to store your email, both new and old. This could be done in many forms, but since SQLite is well known and already in most new browsers and bundled in many operating systems, why not use that? Here's where the problem lies.

We have been talking about the issues with a single-threaded browser. Now imagine operations such as writing new messages to the database or performing long queries. We can't freeze the UI while the database does its work—the latency could be enormous! The Gears team needed a way to get around this. Since the Gears plug-in can do whatever it wants, it can easily work around the lack of threads in JavaScript. But since the need for concurrency is a general problem, why not give this ability to the outside world? Hence the "Worker Pool" API, which led to the HTML 5 standard "Web Workers."

The two APIs look subtly different, but this is because Web Workers is sort of like version 2.0 of the pioneering Gears API; Gears should support the standard API soon. There are already "shim" libraries that bridge the existing Gears API and the standard Web Worker API, and these shims can be used to work even without Gears or Web Workers (by using `setTimeout()`, described in this chapter).

Timers

Another approach, common before Gears and Web Workers, was simply to split up long-running operations into separate chunks and use JavaScript timers to control the execution. For example:

```
var functionState = {};

function expensiveOperation() {
    var startTime = new Date().getMilliseconds();
    while ((new Date().getMilliseconds() - startTime) < 100) {
        // TODO: implement expensive operation in such a way
        // that it performs work in iterative chunks that complete
        // in less than 100 ms and shove state in "functionState"
        // outside this function; good luck with that ;-)
    }

    if (!functionState.isFinished) {
        // re-enter expensiveOperation 10 ms after exiting; experiment
        // with larger values to strike the right balance between UI
        // responsiveness and performance
        setTimeout(expensiveOperation(), 10);
    }
}
```

Splitting up the operation in the manner just illustrated will result in a responsive interface, but as the comment in the listing indicates, it may not be straightforward (or even feasible) to structure the operation in that way. See "Yielding Using Timers" on page 103 for more details on using `setTimeout()` in this manner.

There's another fundamental issue with this approach. Most modern computers have multiple "cores," which means that they have the ability to execute multiple threads in a truly concurrent fashion (whereas previous computers have only emulated concurrency through fast task switching). Implementing task switching manually via JavaScript as we've done in the listing can't take advantage of such architectures; you are therefore leaving processing power on the table by forcing one of the cores to do all of the processing.

Thus, it is possible to perform long-running operations on the browser's main thread and maintain a responsive interface, but it's easier and more efficient to use workers.

XMLHttpRequest

A discussion of threading wouldn't be complete without touching briefly on the famed enabler of the Ajax revolution: `XMLHttpRequest`, or "XHR" for short. Using XHR, a web page may send a message and receive a response entirely from the JavaScript environment, a feat that enables rich interactivity without loading new pages.

XHR has two basic execution modes: synchronous and asynchronous. In the asynchronous mode, XHR is essentially a Web Worker but with a specialized API; indeed, coupled with other features of the in-progress HTML 5 specification, you can re-create the functionality of XHR with a worker. In the synchronous mode, XHR acts as though it performs all of its work on the browser's main thread and will therefore introduce user interface latency that lasts as long as XHR takes to send its request and parse the response from the server. Therefore, never use XHR in synchronous mode, as it can lead to unpredictable user interface latency well outside of tolerable ranges.

Effects of Memory Use on Response Time

There's another key aspect to creating responsive web pages: memory management. Like many modern high-level languages that abstract away low-level memory management, most JavaScript runtimes implement garbage collection (or "GC" for short). Garbage collection can be a magical thing, relieving developers from tedious details that feel more like accounting than programming.

However, automatic memory management comes with a cost. All but the most sophisticated of GC implementations "stop the world" when they perform their collections; that is, they freeze the entire runtime (including what we've been calling the main browser JavaScript thread) while they walk the entire "heap" of created objects, searching for those that are no longer being used and are therefore eligible for recycling into unused memory.

For most applications, GC is truly transparent; the runtime is frozen for short enough periods of time that it escapes the user's attention entirely. However, as an application's memory footprint increases in size, the time required to walk through the entire heap searching for objects that are no longer in use grows and can eventually reach levels that a user does notice.

When this occurs, the application begins to be intermittently sluggish on somewhat regular intervals; as the problem gets worse, the entire browser may freeze on these intervals. Both cases lead to a frustrating user experience.

Most modern platforms provide sophisticated tools that enable you to monitor the performance of the runtime's GC process and to view the current set of objects on the heap in order to diagnose GC-related problems. Unfortunately, JavaScript runtimes don't fall into that category. To make matters worse, no tools exist that can inform developers when collections occur or how much time they are spending performing

their work; such tools would be very helpful to verify that observed latency is related to GC.

This tool gap is a serious detriment toward the development of large-scale browser-hosted JavaScript applications. Meanwhile, developers must guess whether GC is responsible for UI delays.

Virtual Memory

There is another danger associated with memory: paging. Operating systems have two classes of memory they make available to applications: physical and virtual. *Physical memory* is mapped to extremely fast RAM chips in the underlying computer; *virtual memory* is mapped to a much slower mass storage device (e.g., hard drive) that makes up for its relative pokiness with much larger available storage space.

If your web page's memory requirements grow sufficiently large, you may force the operating system to start *paging*, an extremely slow process whereby other processes are forced to relinquish their real memory to make room for the browser's increased appetite. The term *paging* is used because all modern operating systems organize memory into individual *pages*, the term used to describe the smallest unit of memory that is mapped to either real or virtual memory. When paging occurs, pages are transferred from real to virtual memory (i.e., from RAM to a hard drive) or vice versa.

The performance degradation caused by paging is a bit different from GC pauses; paging results in a general, pervasive sluggishness whereas GC pauses tend to manifest themselves as discrete, individual pauses that occur in intervals—though the lengths of the pauses grow over time. Regardless of their differences, either one of these problems represents significant threats to your goal of creating a responsive user interface.

Troubleshooting Memory Issues

As we mentioned earlier, we know of no good memory troubleshooting tools for browser-hosted JavaScript applications. The state of the art is to observe the memory footprint of the browser process (see the section "Measuring Memory Use" at *http://blog.pavlov.net/2008/03/11/firefox-3-memory-usage/* for details on how to measure process memory usage in Windows and OS X), and if it grows larger than is tolerable during the course of your application, check whether your code has any opportunities for memory usage optimizations.

Once you've determined that you have a memory problem, you should look for opportunities to clean up after yourself where you haven't yet done so. You can do this in two ways:

- Use the `delete` keyword to remove JavaScript objects that are no longer needed from memory.
- Remove nodes that are no longer necessary from the web page DOM.

The following code listing demonstrates how to perform both of these tasks:

```
var page = { address: "http://some/url" };

page.contents = getContents(page.address);

...

// later, the contents are no longer necessary
delete page.contents;

...

var nodeToDelete = document.getElementById("redundant");

// remove the node from the DOM (which can only be done via
// call to removeChild() from parent node) and
// simultaneously delete the node from memory
delete nodeToDelete.parent.removeChild(nodeToDelete);
```

Obviously, there is significant room for improvement in the area of memory usage optimization for web pages. At Mozilla, we are currently developing tools to address this problem. In fact, by the time you read this, you should be able to find one or more such tools by visiting *http://labs.mozilla.com*.

Summary

Ajax ushered in a new era of long-running, JavaScript-centric web pages. Such web pages are really browser-hosted applications and are subject to the same user interface guidelines of any other application. It is vital that such applications keep the user interface responsive by minimizing operations performed on the main application thread.

Web Workers are a powerful new tool that can be used to offload complex operations that threaten UI responsiveness. The Gears plug-in and JavaScript timers can be used when Web Workers are unavailable.

Poorly managed memory can lead to UI performance problems. While there's a shortage of good tools to troubleshoot memory problems, developers can generally observe browser memory usage and take steps to minimize their application's memory footprint when problems arise. The good news is that memory troubleshooting tools are in development.

CHAPTER 3

Splitting the Initial Payload

The growing adoption of Ajax and DHTML (Dynamic HTML) means today's web pages have more JavaScript and CSS than ever before. Web applications are becoming more like desktop applications, and just like desktop applications, a large percentage of the application code isn't used at startup. Advanced desktop applications have a plug-in architecture that allows for modules to be loaded dynamically, a practice that many Web 2.0 applications could benefit from. In this chapter I show some popular Web 2.0 applications that load too much code upfront, and I discuss approaches for making pages load more dynamically.

Kitchen Sink

Facebook has 14 external scripts totaling 786 KB uncompressed.* Figuring out how much of that JavaScript is necessary for the initial page to render is difficult to do, even for a core Facebook frontend engineer. Some of those 14 external scripts are critical to rendering the initial page, but others were included because they support Ajax and DHTML functionality, such as the drop-down menus and the Comment and Like features shown in Figure 3-1.

It's critical to render a web page as quickly as possible. Doing so engages the user and creates a responsive experience for her. Imagine if the Facebook JavaScript could be split into two parts: what's needed to render the initial page, and everything else. Rather than bog down the user's first impression with "everything else," the initial JavaScript download should include only what's necessary for the initial rendering. The remaining JavaScript payload can be loaded later.

* Fourteen scripts are downloaded when logged-in users visit this page. If the user is not logged in, fewer scripts are used.

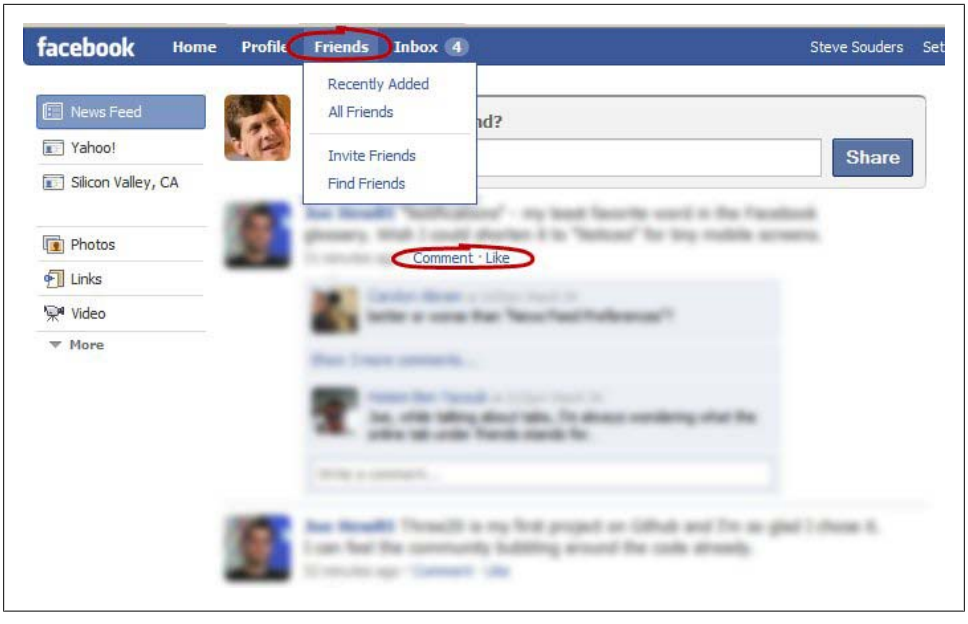


Figure 3-1. Facebook Ajax and DHTML features

This raises several questions:

- How much does this save?
- How do you find where to split the code?
- What about race conditions?
- How do you download "everything else" later?

The first three questions are tackled in this chapter. How to load "everything else" is the topic of Chapter 4.

Savings from Splitting

It turns out that Facebook executes only 9% of the downloaded JavaScript functions by the time the `onload` event is called. This is computed by using Firebug's JavaScript profiler to count all the functions executed up to the `onload` event.† The counting stops at the `onload` event because functionality needed after this point can, and should, be downloaded after the initial page has rendered. I call this a *post-onload download.* (See Chapter 4 for various lazy-loading techniques.)

Table 3-1 shows the percentage of functions downloaded that are not executed before the `onload` event for 10 top U.S. web sites. On average, 75% of the functions

† Firebug is the preeminent web development tool, available at *http://getfirebug.com/*.

downloaded are not executed during the initial rendering of the page. Thus, if downloading of these unexecuted functions was deferred, the size of the initial JavaScript download would be dramatically reduced.

Admittedly, the 75% estimate might be exaggerated; some of the unexecuted functions might be required for error handling or other special conditions. The estimate is still useful to illustrate the point that much of the JavaScript downloaded initially could be deferred. The average total amount of JavaScript is 252 KB uncompressed. This percentage is in terms of function count, not size. If we assume a constant function size, 75% represents an average 189 KB that doesn't have to be downloaded until after the `onload` event, making the initial page render more quickly.

Table 3-1. Percentage of JavaScript functions executed before onload

Web site	% of functions not executed	JavaScript size uncompressed
http://www.aol.com/	71%	115 KB
http://www.ebay.com/	56%	183 KB
http://www.facebook.com/	91%	786 KB
http://www.google.com/search?q=flowers	56%	15 KB
http://search.live.com/results.aspx?q=flowers	75%	17 KB
http://www.msn.com/	69%	131 KB
http://www.myspace.com/	87%	297 KB
http://en.wikipedia.org/wiki/Flowers	79%	114 KB
http://www.yahoo.com/	88%	321 KB
http://www.youtube.com/	84%	240 KB

Finding the Split

Firebug's JavaScript profiler shows the names of all the functions that were executed by the time of the `onload` event. This list can be used to manually split the JavaScript code into one file loaded as part of the initial page rendering and another file to be downloaded later. However, because some of the unused functions may still be necessary for error-handling and other conditional code paths, splitting the code into an initial download that is complete without undefined symbols is a challenge. JavaScript's higher-order features, including function scoping and `eval`, make the challenge even more complicated.

Doloto is a system developed by Microsoft Research for automatically splitting JavaScript into clusters. The first cluster contains the functions needed for initializing the web page. The remaining clusters are loaded on demand the first time the missing code needs to execute, or they are lazy-loaded after the initial flurry of JavaScript activity is over. When applied to Gmail, Live Maps, Redfin, MySpace, and Netflix, Doloto re-

duced the initial JavaScript download size by up to 50% and reduced the application load time by 20% to 40%.

Doloto's decisions about where to split the code are based on a training phase and can result in the JavaScript being split into multiple downloads. For many web applications, it is preferable to define a single split at the `onload` event, after which the remaining JavaScript is immediately downloaded using the nonblocking techniques described in Chapter 4. Waiting to start the additional downloads on demand after the user has pulled down a menu or clicked on a page element forces the user to wait for the additional JavaScript to arrive. This wait can be avoided if all the additional JavaScript is downloaded after the initial page rendering. Until Doloto or other systems are publicly available, developers need to split their code manually. The following section discusses some of the issues to keep in mind when doing this.

Undefined Symbols and Race Conditions

The challenge in splitting your JavaScript code is to avoid undefined symbols. This problem arises if the JavaScript being executed references a symbol that has, mistakenly, been relegated to a later download. In the Facebook example, for instance, I suggest that the JavaScript for drop-down menus should be loaded later. But if the drop-down menu is displayed before the required JavaScript is downloaded, there's a window in which the user can click on the drop-down menu and the required JavaScript won't be available. My suggestion would then have created a race condition where the JavaScript is racing to download while the user is racing to click the menu. In most cases, the JavaScript will win the race, but there is a definite possibility that the user may click first and experience an undefined symbol error when the (yet to be downloaded) drop-down menu function is called.

In a situation where the delayed code is associated with a UI element, the problem can be avoided by changing the element's appearance. In this case, the menu could contain a "Loading..." spinner, alerting the user that the functionality is not yet available.

Another option is to attach handlers to UI elements in the lazy-loaded code. In this example, the menu would be rendered initially as static text. Clicking on it would not execute any JavaScript. The lazy-loaded code would both contain the menu functionality and would attach that behavior to the menu using `attachEvent` in Internet Explorer and `addEventListener` in all other browsers.‡

In situations where the delayed code is not associated with a UI element, the solution to this problem is to use stub functions. A *stub function* is a function with the same name as the original function but with an empty function body or temporary code in place of the original. The previous section described Doloto's ability to download additional JavaScript modules on demand. Doloto implements this on-demand feature

‡ See *http://www.quirksmode.org/js/events_advanced.html* for more information.

by inserting stub functions in the initial download that, when invoked, dynamically download additional JavaScript code. When the additional JavaScript code is downloaded, the original function definitions overwrite the stub functions.

A simpler approach is to include an empty stub function for each function that is referenced but relegated to the later download. If necessary, the stub function should return a stub value, such as an empty string. If the user tries to invoke a DHTML feature before the full function implementation is downloaded, nothing happens. A slightly more advanced solution has each stub function record the user's requests and invokes those actions when the lazy-loaded JavaScript arrives.

Case Study: Google Calendar

A good example of splitting the initial payload is Google Calendar. Figure 3-2 shows the HTTP requests that are made when Google Calendar is requested. I call these charts *HTTP waterfall charts*. Each horizontal bar represents one request. The resource type is shown on the left. The horizontal axis represents time, so the placement of the bars shows at what point during page load each resource was requested and received.

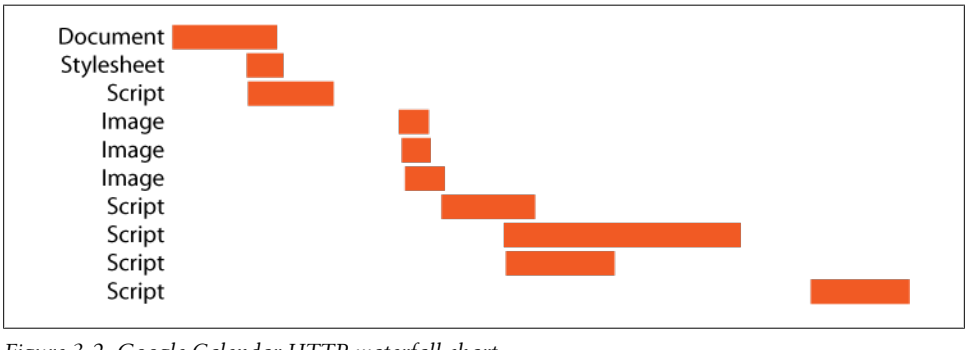


Figure 3-2. Google Calendar HTTP waterfall chart

Google Calendar requests five scripts totaling 330 KB uncompressed. The payload is split into an initial script of 152 KB that is requested early (the third bar from the top). The blocking behavior of this script is mitigated by the fact that it contains less than half of the total JavaScript. The rest of the JavaScript payload is requested last, after the page has been allowed to render.

By splitting their JavaScript, the Google Calendar team creates a page that renders more quickly than it would have if all of the JavaScript were loaded in one file. Splitting a web application's JavaScript is not a simple task. It requires determining the functions needed for initial rendering, finding all required code dependencies, stubbing out other functions, and lazy-loading the remaining JavaScript. Further automation for these tasks is needed. Microsoft's Doloto project describes such a system, but as of this writing, it's not available publicly. Until tools such as this are made available, developers will have to roll up their sleeves and do the heavy lifting themselves.

This chapter has focused on splitting JavaScript, but splitting CSS stylesheets is also beneficial. The savings are less than those gained by splitting JavaScript because the total size of stylesheets is typically less than JavaScript, and downloading CSS does not have the blocking characteristics that downloading JavaScript has.[§] This is another opportunity for further research and tool development.

[§] Firefox 2 is the one exception.

CHAPTER 4

Loading Scripts Without Blocking

SCRIPT tags have a negative impact on page performance because of their blocking behavior. While scripts are being downloaded and executed, most browsers won't download anything else. There are times when it's necessary to have this blocking, but it's important to identify situations when JavaScript can be loaded independent of the rest of the page.

When these opportunities arise, we want to load the JavaScript in such a way that it does not block other downloads. Luckily, there are several techniques for doing this that make pages load faster. This chapter explains these techniques, compares how they affect the browser and performance, and describes the circumstances that make one approach preferred over another.

Scripts Block

JavaScript is included in a web page as an inline script or an external script. An *inline script* includes all the JavaScript in the HTML document itself using the SCRIPT tag:

```
<script>
function displayMessage(msg) {
    alert(msg);
}
</script>
```

External scripts pull in the JavaScript from a separate file using the SCRIPT SRC attribute:

```
<script src='A.js'></script>
```

The SRC attribute specifies the URL of the external file that needs to be loaded. The browser reads the script file from the cache, if available, or makes an HTTP request to fetch the script.

Normally, most browsers download components in parallel, but that's not the case for external scripts. When the browser starts downloading an external script, it won't start any additional downloads until the script has been completely downloaded, parsed, and executed. (Any downloads that were already in progress are not blocked.)

Figure 4-1 shows the HTTP requests for the Scripts Block Downloads example.*

Scripts Block Downloads
http://stevesouders.com/cuzillion/?ex=10008&title=Scripts+Block+Downloads

This page has two scripts at the top, *A.js* and *B.js*, followed by an image, a stylesheet, and an iframe. The scripts are each programmed to take one second to download and one second to execute. The white gaps in the HTTP profile indicate where the scripts are executed. This shows that while scripts are being downloaded and executed, all other downloads are blocked. Only after the scripts have finished are the image, stylesheet, and iframe merrily downloaded in parallel.

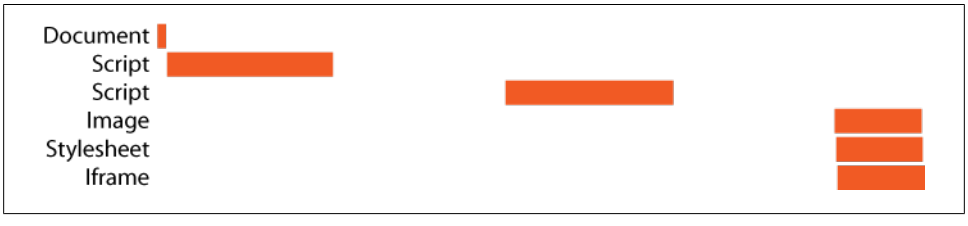


Figure 4-1. Scripts block parallel downloads

The reason browsers block while downloading and executing a script is that the script may make changes to the page or JavaScript namespace that affect whatever follows. The typical example cited is when *A.js* uses `document.write` to alter the page. Another example is when *A.js* is a prerequisite for *B.js*. The developer is guaranteed that scripts are executed in the order in which they appear in the HTML document so that *A.js* is downloaded and executed before *B.js*. Without this guarantee, race conditions could result in JavaScript errors if *B.js* is downloaded and executed before *A.js*.

Although it's clear that scripts must be *executed* sequentially, there's no reason they have to be *downloaded* sequentially. That's where Internet Explorer 8 comes in. The behavior shown in Figure 4-1 is true for most browsers, including Firefox 3.0 and earlier and Internet Explorer 7 and earlier. However, Internet Explorer 8's download profile, shown in Figure 4-2, is different. Internet Explorer 8 is the first browser that supports downloading scripts in parallel.

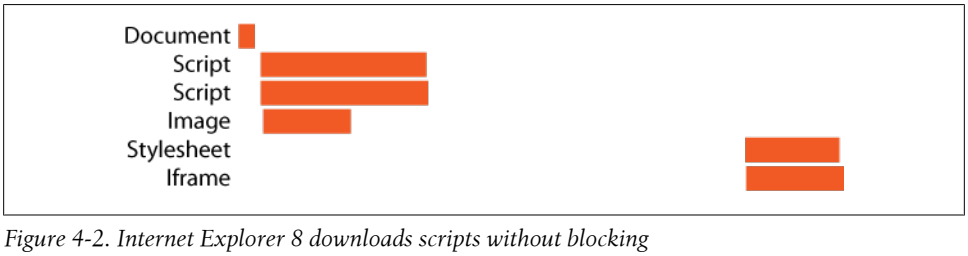


Figure 4-2. Internet Explorer 8 downloads scripts without blocking

* This and other examples are generated from Cuzillion, a tool I built specifically for this chapter. See the Appendix for more information about Cuzillion.

The ability of Internet Explorer 8 to download scripts in parallel makes pages load faster, but as shown in Figure 4-2, it doesn't entirely solve the blocking problem. It is true that *A.js* and *B.js* are downloaded in parallel, but the image and iframe are still blocked until the scripts are downloaded and executed. Safari 4 and Chrome 2 are similar—they download scripts in parallel, but block other resources that follow.†

What we really want is to have scripts downloaded in parallel with all the other components in the page. And we want this in all browsers. The techniques discussed in the next section explain how to do just that.

Making Scripts Play Nice

There are several techniques for downloading external scripts without having your page suffer from their blocking behavior. One technique I *don't* suggest doing is inlining all of your JavaScript. In a few situations (home pages, small amounts of JavaScript), inlining your JavaScript is acceptable, but generally it's better to serve your JavaScript in external files because of the page size and caching benefits derived. (For more information about these trade-offs, see *High Performance Web Sites*, "Rule 8: Make JavaScript and CSS External.")

The techniques listed here provide the benefits of external scripts without the slowdowns imposed from blocking:

- XHR Eval
- XHR Injection
- Script in Iframe
- Script DOM Element
- Script Defer
- `document.write` Script Tag

The following sections describe each of these techniques in more detail, followed by a comparison of how they affect the browser and which technique is best under different circumstances.

XHR Eval

In this technique, an `XMLHttpRequest` (XHR) retrieves the JavaScript from the server. When the response is complete, the content is executed using the `eval` command as shown in this example page.

† As of this writing, Firefox does not yet support parallel script downloads, but that is expected soon.

XHR Eval
http://stevesouders.com/cuzillion/?ex=10009&title=Load+Scripts+using+XHR+Eval

As you can see in the HTTP profile in Figure 4-3, the `XMLHttpRequest` doesn't block the other components in the page—all five resources are downloaded in parallel. The scripts are executed after they finish downloading. (This execution time doesn't show up on the HTTP waterfall chart because no network activity is involved.)

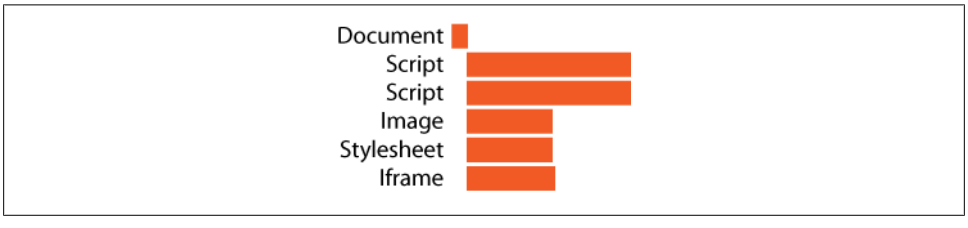


Figure 4-3. Loading scripts using XHR Eval

The main drawback of this approach is that the `XMLHttpRequest` must be served from the same domain as the main page. The relevant source code from the XHR Eval example follows:‡

```
var xhrObj = getXHRObject();
xhrObj.onreadystatechange =
    function() {
        if ( xhrObj.readyState == 4 && 200 == xhrObj.status ) {
            eval(xhrObj.responseText);
        }
    };
xhrObj.open('GET', 'A.js', true); // must be same domain as main page
xhrObj.send('');

function getXHRObject() {
    var xhrObj = false;
    try {
        xhrObj = new XMLHttpRequest();
    }
    catch(e){
        var progid = ['MSXML2.XMLHTTP.5.0', 'MSXML2.XMLHTTP.4.0',
'MSXML2.XMLHTTP.3.0', 'MSXML2.XMLHTTP', 'Microsoft.XMLHTTP'];
        for ( var i=0; i < progid.length; ++i ) {
            try {
                xhrObj = new ActiveXObject(progid[i]);
            }
            catch(e) {
                continue;
            }
```

‡ If you're using a JavaScript library, it probably has a wrapper for `XMLHttpRequest`, such as `jQuery.ajax` or `dojo.xhrGet`. Use that instead of writing your own wrapper.

```
                break;
            }
        }
        finally {
            return xhrObj;
        }
    }
```

XHR Injection

Like XHR Eval, the XHR Injection technique uses an `XMLHttpRequest` to retrieve the JavaScript. But instead of using `eval`, the JavaScript is executed by creating a script DOM element and injecting the `XMLHttpRequest` response into the script. Using `eval` is potentially slower than using this mechanism.

XHR Injection
http://stevesouders.com/cuzillion/?ex=10015&title=XHR+Injection

The `XMLHttpRequest` must be served from the same domain as the main page. The relevant source code from the XHR Injection example follows:

```
var xhrObj = getXHRObject(); // defined in the previous example
xhrObj.onreadystatechange =
    function() {
        if ( xhrObj.readyState == 4 ) {
            var scriptElem = document.createElement('script');
            document.getElementsByTagName('head')[0].appendChild(scriptElem);
            scriptElem.text = xhrObj.responseText;
        }
    };
xhrObj.open('GET', 'A.js', true); // must be same domain as main page
xhrObj.send('');
```

Script in Iframe

Iframes are loaded in parallel with other components in the main page. Whereas iframes are typically used to include one HTML page within another, the Script in Iframe technique leverages them to load JavaScript without blocking, as shown by the Script in Iframe example.

Script in Iframe
http://stevesouders.com/cuzillion/?ex=10012&title=Script+in+Iframe

The implementation is done entirely in HTML:

```
<iframe src='A.html' width=0 height=0 frameborder=0 id=frame1></iframe>
```

Note that this technique uses *A.html* instead of *A.js*. This is necessary because the iframe expects an HTML document to be returned. All that is needed is to convert the external script to an inline script within an HTML document.

Similar to the XHR Eval and XHR Injection approaches, this technique requires that the iframe URL be served from the same domain as the main page. (Browser cross-site security restrictions prevent JavaScript access from an iframe to a cross-domain parent and vice versa.) Even when the main page and iframe are served from the same domain, it's still necessary to modify your JavaScript to create a connection between them. One approach is to have the parent reference JavaScript symbols in the iframe via the `frames` array or `document.getElementById`:

```
// access the iframe from the main page using "frames"
window.frames[0].createNewDiv();

// access the iframe from the main page using "getElementById"
document.getElementById('frame1').contentWindow.createNewDiv();
```

The iframe references its parent using the `parent` variable:

```
// access the main page from within the iframe using "parent"
function createNewDiv() {
    var newDiv = parent.document.createElement('div');
    parent.document.body.appendChild(newDiv);
}
```

Iframes also have an innate cost. In fact, they're the most expensive DOM element by at least an order of magnitude, as discussed in Chapter 13.

Script DOM Element

Rather than using the `SCRIPT` tag in HTML to download a script file, this technique uses JavaScript to create a script DOM element and set the `SRC` property dynamically. This can be done with just a few lines of JavaScript:

```
var scriptElem = document.createElement('script');
scriptElem.src = 'http://anydomain.com/A.js';
document.getElementsByTagName('head')[0].appendChild(scriptElem);
```

Creating a script this way does not block other components during download. As opposed to the previous techniques, Script DOM Element allows you to fetch the JavaScript from a server other than the one used to fetch the main page. The code to implement this technique is short and simple. Your external script file can be used as is and doesn't need to be refactored as in the XHR Eval and Script in Iframe approaches.

Script DOM Element
http://stevesouders.com/cuzillion/?ex=10010&title=Script+Dom+Element

Script Defer

Internet Explorer supports the `SCRIPT DEFER` attribute as a way for developers to tell the browser that the script does not need to be loaded immediately. This is a safe attribute to use when a script does not contain calls to `document.write` and no other scripts in

the page depend on it. When Internet Explorer downloads the deferred script, it allows other downloads to be done in parallel.

Script Defer
http://stevesouders.com/cuzillion/?ex=10013&title=Script+Defer

The `DEFER` attribute is an easy way to avoid the bad blocking behavior of scripts with the addition of a single word:

```
<script defer src='A.js'></script>
```

Although `DEFER` is part of the HTML 4 specification, it is implemented only in Internet Explorer and in some newer browsers.

document.write Script Tag

This last technique uses `document.write` to put the `SCRIPT` HTML tag into the page.

`document.write` Script Tag
http://stevesouders.com/cuzillion/?ex=10014&title=document.write+Script+Tag

This technique, similar to Script Defer, results in parallel script loading in Internet Explorer only. Although it allows multiple scripts to be downloaded in parallel (provided all the `document.write` lines occur in the same script block), other types of resources remain blocked while scripts are downloading:

```
document.write("<script type='text/javascript' src='A.js'><\/script>");
```

Browser Busy Indicators

All of the techniques described in the preceding section improve how JavaScript is downloaded by allowing multiple resources to be downloaded in parallel. But these techniques differ in certain other aspects. One area of differentiation is how they affect the user's perception of whether the page is loaded. Browsers offer multiple *browser busy indicators* that give the user clues that the page is still loading.

Figure 4-4 shows four browser busy indicators: the status bar, the progress bar, the tab icon, and the cursor. The status bar shows the URL of the current download. The progress bar moves across the bottom of the window as downloads complete. The logo spins while downloads are happening. The cursor changes to an hourglass or similar cursor to indicate that the page is busy.

The other two browser busy indicators are blocked rendering and blocked `onload` event. Blocked rendering is very obtrusive to the user experience. When scripts are being downloaded in the typical manner using `SCRIPT SRC`, nothing below the script is rendered. Freezing the page before it's fully rendered is a severe way of showing the browser is busy.

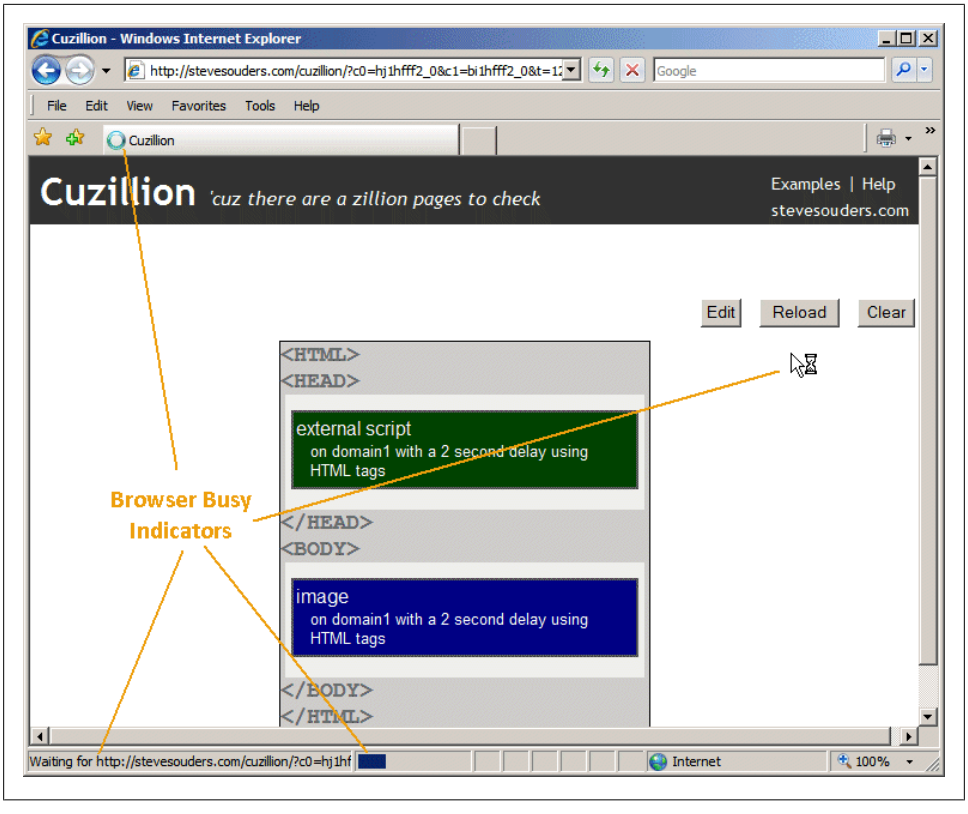


Figure 4-4. Busy indicators in the browser

Typically, the page's `onload` event doesn't fire until all resources have been downloaded. This may affect the user experience if the status bar takes longer to say "Done" and setting focus on the default input field is delayed.

Whereas most of these browser busy indicators are triggered when downloading JavaScript in the usual `SCRIPT SRC` way, none of them are triggered by the XHR Eval and XHR Injection techniques when using Internet Explorer, Firefox, and Opera. The busy indicators that are triggered vary depending on the technique used and the browser being tested.

Table 4-1 shows which busy indicators occur for each of the JavaScript download techniques. XHR Eval and XHR Injection trigger the fewest busy indicators. The other techniques have mixed behavior. Although busy indicators vary across browsers, they're generally consistent across different browser versions.

Table 4-1. Browser busy indicators triggered by JavaScript downloads

Technique	Status bar	Progress bar	Logo	Cursor	Block render	Block onload
Normal Script Src	FF, Saf, Chr	IE, FF, Saf	IE, FF, Saf, Chr	FF, Chr	IE, FF, Saf, Chr, Op	IE, FF, Saf, Chr, Op
XHR Eval	Saf, Chr	Saf	Saf, Chr	Saf, Chr	--	--
XHR Injection	Saf, Chr	Saf	Saf, Chr	Saf, Chr	--	--
Script in Iframe	IE, FF, Saf, Chr	FF, Saf	IE, FF, Saf, Chr	FF, Chr	--	IE, FF, Saf, Chr, Op
Script DOM Element	FF, Saf, Chr	FF, Saf	FF, Saf, Chr	FF, Chr	--	FF, Saf, Chr
Script Defer[a]	FF, Saf, Chr	FF, Saf	FF, Saf, Chr	FF, Chr, Op	FF, Saf, Chr, Op	IE, FF, Saf, Chr, Op
`document.write` Script Tag[b]	FF, Saf, Chr	IE, FF, Saf	IE, FF, Saf, Chr	FF, Chr, Op	IE, FF, Saf, Chr, Op	IE, FF, Saf, Chr, Op

[a] Script Defer achieves parallel downloads in Firefox 3.1 and later.

[b] Note that `document.write` Script Tag achieves parallel downloads only in Internet Explorer, Safari 4, and Chrome 2.

Abbreviations are as follows: (Chr) Chrome 1.0.154 and 2.0.156; (FF) Firefox 2.0, 3.0, and 3.1; (IE) Internet Explorer 6, 7, and 8; (Op) Opera 9.63 and 10.00 alpha; (Saf) Safari 3.2.1 and 4.0 (developer preview).

It's important to understand how each technique behaves with regard to the browser busy indicators. In some cases, the busy indicators are desirable for a better user experience: they let the user know the page is working. In other situations, it would be better not to show any busy activity, thus encouraging users to start interacting with the page.

Ensuring (or Avoiding) Ordered Execution

In many cases, a web page contains multiple scripts that have a particular dependency order. Using the normal `SCRIPT SRC` approach guarantees that the scripts are downloaded and executed in the order in which they are listed in the page. However, using certain of the advanced downloading techniques described previously does not carry such a guarantee. Because the scripts are downloaded in parallel, they may get executed in the order in which they arrive—the fastest response to arrive being executed first—rather than the order in which they were listed. This can lead to race conditions resulting in undefined symbol errors.

Some of the techniques do ensure ordered execution, but they vary depending on the browser. For Internet Explorer, the Script Defer and `document.write` Script Tag

approaches that guarantee scripts are executed in the order listed, regardless of which is downloaded first. For instance, the IE Ensure Ordered Execution example contains three scripts that are loaded using Script Defer. Even though the first script (with `sleep=3` in the URL) finishes downloading last, it is still the first to be executed.

IE Ensure Ordered Execution
: *http://stevesouders.com/cuzillion/?ex=10017&title=IE+Ensure+Ordered+Execution*

Because the Script Defer and `document.write` Script Tag techniques don't achieve parallel script downloads in Firefox, you need to use a different technique whenever one script depends on another. The Script DOM Element approach guarantees that scripts are executed in the order listed in Firefox. The FF Ensure Ordered Execution example contains three scripts that are loaded using the Script DOM Element approach. Even though the first script (with `sleep=3` in the URL) finishes downloading last, it is still the first to be executed.

FF Ensure Ordered Execution
: *http://stevesouders.com/cuzillion/?ex=10018&title=FF+Ensure+Ordered+Execution*

It's not always important to ensure that scripts are executed in the order specified. Sometimes you actually want the browser to execute whatever script happens to come first, because that produces a page that loads faster. One example is a web page containing multiple widgets (A, B, and C) with associated scripts (*A.js*, *B.js*, and *C.js*) that do not have any interdependencies. Even though the page might list the widget scripts in that order, a better user experience would result from executing whichever widget script is received first. The XHR Eval and XHR Injection techniques achieve this. The Avoid Ordered Execution example executes the first script downloaded, even though it's not the first script listed in the page.

Avoid Ordered Execution
: *http://stevesouders.com/cuzillion/?ex=10019&title=Avoid+Ordered+Execution*

Summarizing the Results

I've presented several advanced techniques for downloading external scripts and various trade-offs between them. Table 4-2 summarizes the results.

Table 4-2. Summary of advanced script downloading techniques

Technique	Parallel downloads	Domains can differ	Existing scripts	Busy indicators	Ensures order	Size (bytes)
Normal Script Src	(IE8, Saf4)[a]	Yes	Yes	IE, Saf4, (FF, Chr)[b]	IE, Saf4, (FF, Chr, Op)[c]	~50

Technique	Parallel downloads	Domains can differ	Existing scripts	Busy indicators	Ensures order	Size (bytes)
XHR Eval	IE, FF, Saf, Chr, Op	No	No	Saf, Chr	--	~500
XHR Injection	IE, FF, Saf, Chr, Op	No	Yes	Saf, Chr	--	~500
Script in Iframe	IE, FF, Saf, Chr, Op[d]	No	No	IE, FF, Saf, Chr	--	~50
Script DOM Element	IE, FF, Saf, Chr, Op	Yes	Yes	FF, Saf, Chr	FF, Op	~200
Script Defer	IE, Saf4, Chr2, FF3.1	Yes	Yes	IE, FF, Saf, Chr, Op	IE, FF, Saf, Chr, Op	~50
`document.write` Script Tag	(IE, Saf4, Chr2, Op)[e]	Yes	Yes	IE, FF, Saf, Chr, Op	IE, FF, Saf, Chr, Op	~100

[a] Scripts are downloaded in parallel with other scripts, but other types of resources are blocked from downloading.

[b] These browsers do not, however, support parallel downloads with this technique.

[c] See note a above.

[d] An interesting performance boost in Opera is that in addition to the script iframes being downloaded in parallel, the code is executed in parallel, too.

[e] See note b above.

Abbreviations are as follows: (Chr) Chrome 1.0.154 and 2.0.156; (FF) Firefox 2.0 and 3.1; (IE) Internet Explorer 6, 7, and 8; (Op) Opera 9.63 and 10.00 alpha; (Saf) Safari 3.2.1 and 4.0 (developer preview).

These techniques allow scripts to be downloaded in parallel with all the other resources in the page, something that browsers don't do by default, even newer browsers. This can significantly speed up your web page. This is especially important for Web 2.0 applications, where the number and size of external scripts are greater than in other web pages.

The `document.write` Script Tag technique is less preferred because it parallelizes downloads only in a subset of browsers and blocks parallel downloads for anything other than script resources. Script Defer also parallelizes downloads in only some browsers.

XHR Eval, XHR Injection, and Script in Iframe carry the requirement that your scripts reside on the same hostname as the main page. To use the XHR Eval and Script in Iframe techniques, you must refactor your scripts slightly, whereas the XHR Injection and Script DOM Element approaches can download your existing script files without any changes. An estimate of the number of characters added to the page to implement each technique is shown in the "Size" column in Table 4-2.

The different effects that each technique has on the browser's busy indicators bring in another set of considerations. If you're downloading scripts that are incidental to the initial rendering of the page (i.e., "lazy-loading"), techniques that make the page appear complete are preferred, such as XHR Eval and XHR Injection. If you want to indicate to the user that the page is still loading while the browser downloads scripts, Script in Iframe is better because it triggers more browser busy indicators.

The final issue of ordered execution favors some techniques over others depending on whether load order matters. If you want scripts to be downloaded in parallel with other resources but executed in a specific order, it's necessary to mix techniques by browser. If load order doesn't matter, XHR Eval and XHR Injection can be used.

And the Winner Is

My conclusion is that there is no single best solution. The preferred approach depends on your requirements. Figure 4-5 shows the decision tree for selecting the best technique for downloading scripts.

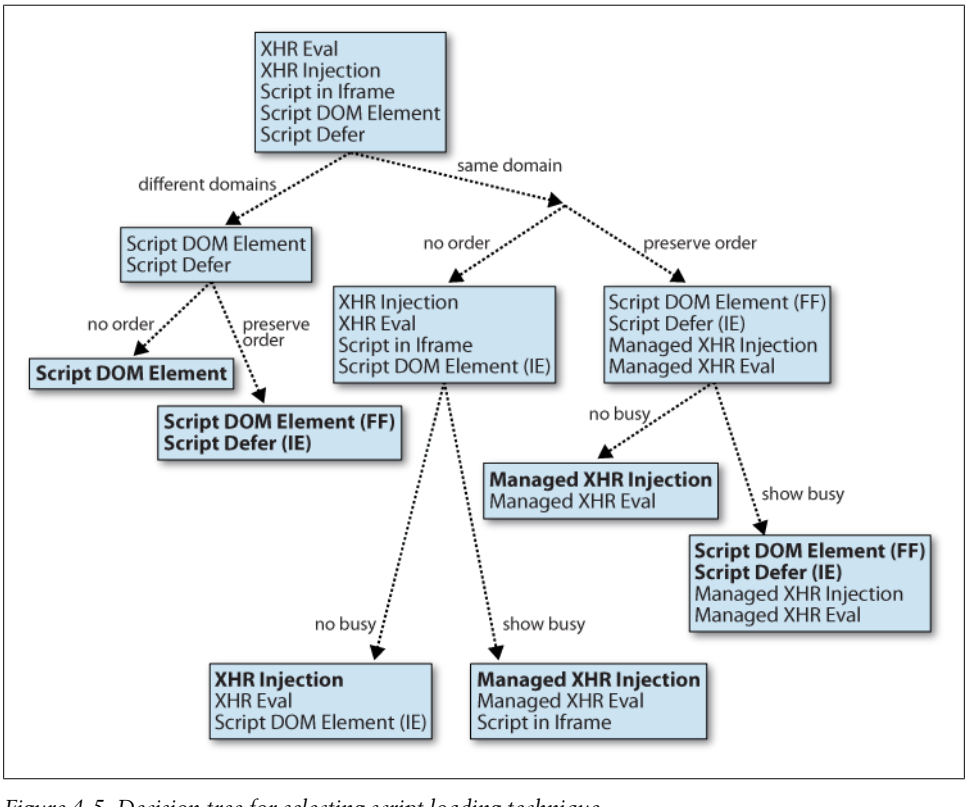


Figure 4-5. Decision tree for selecting script loading technique

There are six possible outcomes in this decision tree:

Different Domains, No Order
: XHR Eval, XHR Injection, and Script in Iframe can't be used under these conditions because the domain of the main page is different from the domain of the script. Script Defer shouldn't be used because it forces scripts to be loaded in order, whereas the page loads faster if scripts are executed as soon as they arrive. For this situation, Script DOM Element is the best alternative. In Firefox, load order is preserved even though that's not desired. Note that both of these techniques trigger the busy indicators, so there's no way to avoid that. Examples of web pages that match this situation are pages that contain JavaScript-enabled ads and widgets. The scripts for these ads and widgets are likely on domains that differ from the main page, but they don't have any interdependencies, so load order doesn't matter.

Different Domains, Preserve Order
: As before, because the domains of the main page and scripts are different, XHR Eval, XHR Injection, and Script in Iframe are not viable alternatives. To ensure load order, Script Defer should be used for Internet Explorer and Script DOM Element for Firefox. Note that both of these techniques trigger the busy indicators. An example of a page that matches these requirements is a page pulling in multiple JavaScript files from different servers that have interdependencies.

Same Domain, No Order, No Busy Indicators
: XHR Eval and XHR Injection are the only techniques that do not trigger the busy indicators. Of the two XHR techniques, I prefer XHR Injection because it can be used without refactoring the existing scripts. This technique would apply to a web page that wanted to download its own JavaScript file in the background, as described in Chapter 3.

Same Domain, No Order, Show Busy Indicators
: XHR Eval, XHR Injection, and Script in Iframe are the only techniques that do not preserve load order across both Internet Explorer and Firefox. Script in Iframe seems to be the best choice because it triggers the busy indicators and increases the size of the page only slightly, but I prefer XHR Injection because it can be used without any refactoring of the existing scripts and it's already a choice for other decision tree outcomes. Additional client-side JavaScript is required to activate the busy indicators: the status bar and cursor can be activated when the XHR is sent and then deactivated when the XHR returns. I call this "Managed XHR Injection."

Same Domain, Preserve Order, No Busy Indicators
: XHR Eval and XHR Injection are the only techniques that do not trigger the busy indicators. Of the two XHR techniques, I prefer XHR Injection because it can be used without refactoring the existing scripts. To preserve load order, another type of "Managed XHR Injection" is needed. In this case, the XHR responses are queued if necessary to handle the situation where a script that needs to be loaded later in the order is not executed until all the preceding scripts have been downloaded

and executed. An example of a page in this situation is one where multiple interdependent scripts need to be downloaded in the background.

Same Domain, Preserve Order, Show Busy Indicators
: Script Defer for Internet Explorer and Script DOM Element for Firefox are the preferred solutions here. Managed XHR Injection and Managed XHR Eval are other valid alternatives, but they add more code to the main page and are more complicated to implement.

The next step is to implement this logic in code by providing a simple function that developers can use to make sure they load scripts in the optimal way. A prototype for such a function would look like this:

```
function loadScript(url, bPreserveOrder, bShowBusy);
```

To avoid downloading more JavaScript than necessary, a backend implementation in a language invoked by the server, such as Perl, PHP, or Python, would be the most useful. In their backend templates, web developers would call this function and the appropriate technique would be inserted into the HTML document response. Providing support for these advanced best practices in development frameworks is the appropriate next step for getting wider adoption.

CHAPTER 5
Coupling Asynchronous Scripts

Chapter 4 explains how to load external scripts asynchronously. When scripts are loaded the normal way (`<script src="url"></script>`), they block all other downloads in the page, and any elements below the script are blocked from rendering. Loading scripts asynchronously avoids this blocking behavior, resulting in a page that loads and feels faster.

The performance benefit of loading scripts without blocking comes at a cost. Whenever code is executed asynchronously, race conditions are possible. In the case of external scripts, the concern is inline scripts that use symbols defined in the external script. If the external script is loaded asynchronously without thought to the inlined code, race conditions may result in undefined symbol errors.

When there is a code dependency between an asynchronously loaded external script and an inline script, the two scripts have to be coupled in such a way as to guarantee execution order. Not surprisingly, there's no easy way to do this across all browsers. The problem and several solutions are presented in this chapter, broken down into the following sections:

Code Example: menu.js

Ensuring execution order was one of the traits discussed in Chapter 4. That discussion focused on the execution order of external scripts, but most web pages that load external scripts also include inline scripts that use the external script's symbols, such as pages that use Google Analytics and popular JavaScript frameworks such as jQuery and the Yahoo! UI Library.

To illustrate this situation, I created the Normal Script Src example that has an external script followed by an inline script with code dependencies. The external script, *menu.js*, provides functionality to draw a drop-down menu, as shown in Figure 5-1.

Normal Script Src
http://stevesouders.com/efws/couple-normal.php

The Normal Script Src implementation, shown in the following code sample, starts by loading *menu.js* in the normal way. The inline script that follows creates the menu. The inline script defines `aExamples`, an array of menu items. The `init` function calls `EFWS.Menu.createMenu`, passing in an element ID (`'examplesbtn'`) and the array of menu items. The `'examplesbtn'` element is what the menu is attached to—in this case, the button in the page labeled "Examples":

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

<script type="text/javascript">
var aExamples =
    [
     ['couple-normal.php', 'Normal Script Src'],
     ['couple-xhr-eval.php', 'XHR Eval'],
     ...
    ];


function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}
```

```
init();
</script>
```

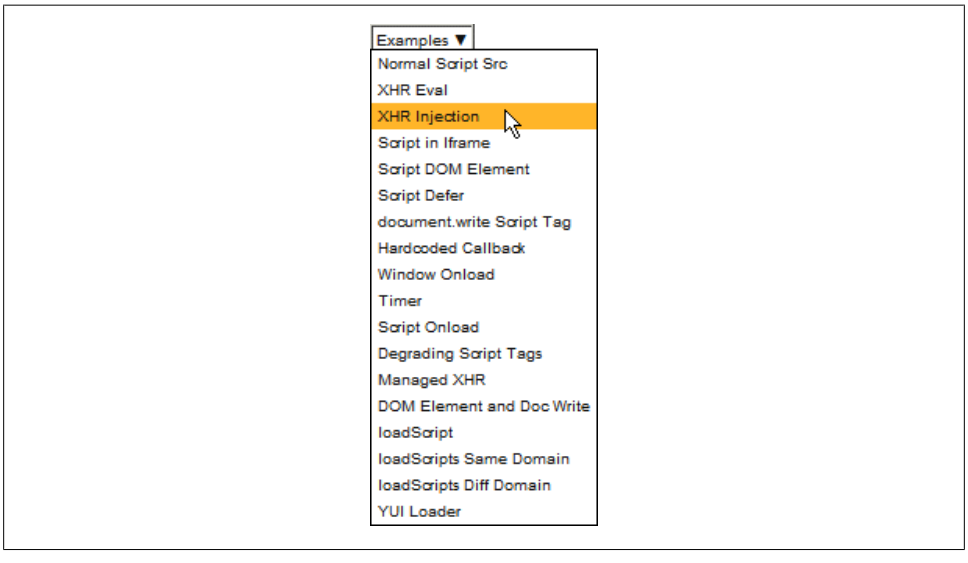


Figure 5-1. menu.js example

The *menu.js* example is the scenario that motivates this chapter and that is encountered in many web sites today. The page contains an external script and an inline script. The inline script depends on the external script, so it's critical that the execution order be preserved—the external script must be downloaded, parsed, and executed before the inline script.

In addition, this is a perfect opportunity for asynchronous script loading so that the downloading of other resources in the page isn't blocked. The "other resources" in this example page is an image. The image is configured to take one second to download, while *menu.js* takes two seconds. If the script is loaded the normal way, the image is blocked from downloading, as shown in Figure 5-2.

Figure 5-2. Normal Script Src HTTP waterfall chart

If *menu.js* was loaded asynchronously, the image wouldn't be blocked and the page would load faster. Furthermore, *menu.js* is a good candidate for asynchronous loading because it doesn't render any part of the visible page. It provides functionality that is

accessible only after the page has rendered. The question is: can we load *menu.js* asynchronously without triggering any undefined symbol errors in the inline script?

Race Conditions

The Normal Script Src example doesn't produce any undefined symbol errors, but *menu.js* blocks the image download, making the page load more slowly. To improve performance, it would be better to load *menu.js* asynchronously, but we need to determine whether execution order is preserved, or whether a race condition produces undefined symbol errors.

I converted the Normal Script Src example to use the nonblocking techniques from Chapter 4. In each example, I programmatically answer two questions: Was the script loaded without blocking? Was the execution order preserved?

XHR Eval
: *http://stevesouders.com/efws/couple-xhr-eval.php*

XHR Injection
: *http://stevesouders.com/efws/couple-xhr-injection.php*

Script in Iframe
: *http://stevesouders.com/efws/couple-script-iframe.php*

Script DOM Element
: *http://stevesouders.com/efws/couple-script-dom.php*

Script Defer
: *http://stevesouders.com/efws/couple-script-defer.php*

`document.write` Script Tag
: *http://stevesouders.com/efws/couple-doc-write.php*

Table 5-1 shows the results of running these examples across major browsers. None of the techniques perform downloads in parallel while preserving execution order for a specific browser. The one exception is the Script DOM Element approach in Firefox.

Table 5-1. Ensuring execution order for external and inline scripts

Technique	Download script and image in parallel	Ensure execution order
Normal Script Src	IE8, Saf4, Chr2	IE, FF, Saf, Chr, Op
XHR Eval	IE, FF, Saf, Chr, Op	--
XHR Injection	IE, FF, Saf, Chr, Op	--
Script in Iframe	IE, FF, Saf, Chr, Op[a]	--
Script DOM Element	IE, FF, Saf, Chr	FF, Op
Script Defer	IE, (Saf4, Chr2)[b]	FF, Saf, Chr, Op

Technique	Download script and image in parallel	Ensure execution order
`document.write` Script Tag	Saf4, Chr2	IE, FF, Saf, Chr, Op

[a] An interesting performance boost in Opera is that in addition to the script iframes being downloaded in parallel, the code is executed in parallel, too.

[b] In these newer browsers, scripts download in parallel by default. The `DEFER` attribute has no effect.

Abbreviations are as follows: (Chr) Chrome 1.0.154 and 2.0.156; (FF) Firefox 2.0 and 3.1; (IE) Internet Explorer 6, 7, and 8; (Op) Opera 9.63 and 10.00 alpha; (Saf) Safari 3.2.1 and 4.0 (developer preview).

Newer browsers show a brighter future. Internet Explorer 8, Safari 4, and Chrome 2 achieve parallelization and execution order using the normal `SCRIPT` tags (`<script src="`*`url`*`"></script>`). However, scripts loaded in Internet Explorer 8 and Chrome 2 still block certain other resources from loading, such as the image in these test pages. It's also important, perhaps more important, to speed up pages in the mainstream browsers that are still popular, including Internet Explorer 6 and 7. What's needed is a way to load scripts asynchronously *and* preserve execution order across browsers. The coupling techniques described in the following section do just that.

Preserving Order Asynchronously

When external scripts are loaded the normal way, inlined code is blocked from executing and race conditions aren't a concern. Once we start loading scripts asynchronously, one of the techniques presented in this section is needed to couple the inlined code with the external script on which it depends. The coupling techniques are:

- Hardcoded Callback
- Window Onload
- Timer
- Script Onload
- Degrading Script Tags

Script Onload is likely to be your best choice, but I walk through some of the other techniques first in order to highlight the issues.

The coupling examples in this section use the Script DOM Element approach as the asynchronous loading technique, as described in Chapter 4. This approach uses JavaScript to create a script element and set its `SRC` attribute to *menu.js*. The code shown here is taken from the Script DOM Element example:

```
<script type="text/javascript">
var domscript = document.createElement('script');
domscript.src = "menu.js";
document.getElementsByTagName('head')[0].appendChild(domscript);
```

```
</script>

<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

init();
</script>
```

This is my preferred nonblocking technique because it is lightweight and scripts can be loaded from domains that differ from the main page. As shown in Figure 5-3, this technique successfully downloads the external script (two seconds long) in parallel with the image (one second). However, this approach produces undefined symbol errors in Internet Explorer, Safari, and Chrome because the inlined code is executed before the asynchronously loaded script has arrived. The Script DOM Element approach does not preserve order in these three browsers, as confirmed by their absence in the "Ensure execution order" column in Table 5-1. The coupling techniques discussed in the following sections solve these race condition problems.

Figure 5-3. Script DOM Element HTTP waterfall chart

Technique 1: Hardcoded Callback

A simple coupling technique is to have the external script call a function in the inlined code. In our example, this is done by adding a call to `init` at the bottom of the external script (now called *menu-with-init.js*). This approach is demonstrated in the Hardcoded Callback example.

Hardcoded Callback
: *http://stevesouders.com/efws/hardcoded-callback.php*

The inlined code has a few modifications. The call to `init` is removed—that's now being called from the external script. The definitions of `aExamples` and `init` are moved above the insertion of *menu-with-init.js*, so they will be available when *menu-with-init.js* finishes loading:

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}
```

```
var domscript = document.createElement('script');
domscript.src = "menu.js";
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>
```

If the web developer controls both the main page and the external script, this is a viable technique. However, it's not always possible to embed a callback in third-party JavaScript modules. Also, this approach isn't very flexible—changing the callback interface requires coordinating a change in the external script.

Technique 2: Window Onload

This approach kicks off the execution of the inlined code by way of the window's `onload` handler. This preserves execution order as long as the external script is guaranteed to have been downloaded and executed before `window.onload`. Some, but not all, of the asynchronous loading techniques make this guarantee:

- Script in Iframe ensures execution order in Internet Explorer, Firefox, Safari, Chrome, and Opera.
- Script DOM Element ensures execution order in Firefox, Safari, and Chrome.
- Script Defer ensures execution order in Internet Explorer.

Using one of these script loading techniques and coupling the inline script via `window.onload` achieves parallel downloading while preserving execution order, as demonstrated in the Window Onload example.

Window Onload
http://stevesouders.com/efws/window-onload.php

This example uses the Script in Iframe approach to load the external script, since that blocks the `onload` event across most browsers. Instead of loading *menu.js*, the code is embedded in *menu.php* and loaded as an iframe. The inlined code is modified to tie `init` to the window's `onload` event. This is done using either `addEventListener` or `attachEvent`, depending on the browser. This is better than simply doing `window.onload=init` because it ensures that any existing `onload` handlers are not affected:

```
<iframe src="menu.php" width=0 height=0 frameborder=0></iframe>

<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

if ( window.addEventListener ) {
    window.addEventListener("load", init, false);
}
```

```
else if ( window.attachEvent ) {
    window.attachEvent("onload", init);
}
</script>
```

There are two downsides to the Window Onload coupling technique. First, you have to make sure the script is asynchronously loaded in a way that blocks the `onload` event. (That's why I switched from my preferred Script DOM Element technique to Script in Iframe.) Second, the inlined code might be executed later than necessary. If the page contained more resources (images, Flash, etc.), the external script might finish well before the `onload` event fired. Typically, it's preferred to call the inlined code as soon as the external script is finished downloading and executing. In this example, calling the inlined code earlier would make the menu available sooner.

Technique 3: Timer

The Timer technique uses a polling approach to ensure that dependencies are loaded before the inlined code is executed. This is done using `setTimeout` as shown in the Timer example.

Timer
http://stevesouders.com/efws/timer.php

This example's inlined code is modified to include a new function, `initTimer`, which checks whether the required namespace (`EFWS`) exists. If so, `init` is called. If not, `initTimer` is called again after a specified amount of time (300 milliseconds):

```
<script type="text/javascript">
var domscript = document.createElement('script');
domscript.src = "menu.js";
document.getElementsByTagName('head')[0].appendChild(domscript);

var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

function initTimer() {
    if ( "undefined" === typeof(EFWS) ) {
        setTimeout(initTimer, 300);
    }
    else {
        init();
    }
}

initTimer();
</script>
```

If the timer value used in `setTimeout` is too small, this polling technique could add overhead to the page. Conversely, setting it too large will cause an undesirable delay

between when the external script is loaded and when the inlined code is called. One edge case that this simplified code sample doesn't address is when *menu.js* fails to load, in which case the polling will continue indefinitely. Finally, this approach increases maintenance slightly in that a specific symbol from the external script is used to determine when it's done loading. If that symbol changes in the external script, the inlined code would need to be updated.

Technique 4: Script Onload

The previous coupling techniques add brittleness, delays, and overhead to the page. The Script Onload approach addresses all of these issues by attaching to the script's `onload` event.

Script Onload
: *http://stevesouders.com/efws/script-onload.php*

The changes in this example involve the script element's `onload` and `onreadystatechange` handlers. Both are set to call `init`. We prevent `init` from being called twice in Opera by adding the `onloadDone` flag:

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

var domscript = document.createElement('script');
domscript.src = "menu.js";
domscript.onloadDone = false;
domscript.onload = function() {
    domscript.onloadDone = true;
    init();
};
domscript.onreadystatechange = function() {
    if ( "loaded" === domscript.readyState && ! domscript.onloadDone ) {
        domscript.onloadDone = true;
        init();
    }
}
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>
```

Script Onload is the preferred technique for coupling asynchronously loaded external scripts with inline scripts. It doesn't reference any of the symbols in the external script, so maintenance is simpler. The inlined code is executed as early as possible, immediately after the external script is done loading. Using events requires minimal processing.

Technique 5: Degrading Script Tags

This technique is based on John Resig's blog post, "Degrading Script Tags". John is a JavaScript evangelist from Mozilla and the creator of jQuery, the popular JavaScript framework. He describes this technique as a way to couple the jQuery external script with inlined code that accesses the jQuery symbols. This pattern uses one `SCRIPT` tag to include an external script and the inlined code that uses it, like this:

```
<script src="jquery.js" type="text/javascript">
jQuery("p").addClass("pretty");
</script>
```

The idea is that the inlined code is executed after the external script successfully loads. This pattern has several benefits:

Cleaner
: There is one `SCRIPT` tag instead of two.

Clearer
: The inlined code's dependency on the external script is more obvious.

Safer
: If the external script fails to load, the inlined code is not executed, avoiding undefined symbol errors.

There's one downside: today's browsers don't support such syntax! John confirms that browsers load the external script but ignore the inlined code. However, he provides a code sample that shows this can be made to work with a slight addition to the external script. I've applied this technique in the Degrading Script Tags Normal example.

Degrading Script Tags Normal
: *http://stevesouders.com/efws/degrading-script-tag-normal.php*

The inline script follows John's pattern. It uses one `SCRIPT` tag to both specify the external script and inline the dependent code:

```
<script src="menu-degrading.js" type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

init();
</script>
```

The addition of a few lines of JavaScript at the bottom of *menu-degrading.js* is needed to make this work. This new code iterates over all the script elements in the page, searching for the one with a `src` that contains "menu-degrading.js". Basically, the external script is searching for itself in the DOM. When it finds the appropriate script element, it evaluates the script's `innerHTML`:

```
var scripts = document.getElementsByTagName("script");
var cntr = scripts.length;
while ( cntr ) {
    var curScript = scripts[cntr-1];
    if ( -1 != curScript.src.indexOf("menu-degrading.js") ) {
        eval( curScript.innerHTML );
        break;
    }
    cntr--;
}
```

The Degrading Script Tags Normal example works in all the browsers tested: Internet Explorer 6 through 8, Firefox 2 and 3, Safari 3 and 4, Chrome 1 and 2, and Opera 9 and 10. However, the external script is not loaded asynchronously. (Notice that the image isn't loaded until three seconds into the page, instead of the usual one second.) To avoid the blocking behavior of scripts, it's necessary to combine this pattern with one of the asynchronous script loading techniques. I've done this in the Degrading Script Tags Async example.

Degrading Script Tags Async
http://stevesouders.com/efws/degrading-script-tag.php

This example uses the same external script, *menu-degrading.js*, which has the extra code to find the script and evaluate its `innerHTML`. But instead of using the `SCRIPT` tag to pull in the external script, the Script DOM Element nonblocking technique is used. The inlined code is added to the script element dynamically by setting the script element's `text` property (or `innerHTML` in the case of Opera) to `"init();"`:

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
}

var domscript = document.createElement('script');
domscript.src = "menu-degrading.js";
if ( -1 != navigator.userAgent.indexOf("Opera") ) {
    domscript.innerHTML = "init();";
}
else {
    domscript.text = "init();";
}
document.getElementsByTagName('head')[0].appendChild(domscript);
</script>
```

I like this technique for its elegance and simplicity. But this pattern is less well known than Script Onload; it is likely to catch most developers by surprise. It has less overhead (no event handlers are used). It provides a coupling mechanism that is both practical and elegant even when the external script is not loaded asynchronously. The primary drawback is that this technique requires modifying the external script, something that

is not always possible, especially when using third-party JavaScript libraries. At least for now, the Script Onload coupling technique is the best choice.

Multiple External Scripts

The examples so far focus on coupling a single external script with inlined code. This is useful in many situations where the JavaScript framework being used is contained in a single file, such as Google Analytics and jQuery. Often, however, we have *multiple* external scripts and an inline script, all of which must be executed in the order specified. None of the techniques described so far, both here and in Chapter 4, provide a means to preserve order while loading multiple scripts asynchronously. There is no complete solution to this problem, primarily due to browser inconsistencies.

This section describes the two best techniques for loading multiple scripts asynchronously while preserving execution order across the external scripts and inline script. The Managed XHR technique works, but it is restricted to scripts with the same domain as the main page. The DOM Element and Doc Write technique works for scripts on a different domain, but the code varies depending on the User Agent and this technique doesn't load all resource types asynchronously across all browsers.

In order to have an example that uses multiple scripts, I created *menutier.js*. This new script extends the menu functionality to give a tiered or grouped menu, as shown in Figure 5-4 (notice the shaded group headings). In addition, *menutier.js* depends on *menu.js*, so their execution order must be preserved. A tiered menu is created in the inlined code by calling `EFWS.Menu.createTieredMenu`. This sets up the situation we're trying to analyze: multiple external scripts and an inline script that must be executed in order. Furthermore, *menutier.js* is configured to return before *menu.js* on which it depends. Are we headed for trouble? Let's look at how the Managed XHR and DOM Element and Doc Write techniques load the external scripts in parallel while preserving execution order.

Managed XHR

"Managed XHR" is the name used in Chapter 4 for the asynchronous loading technique that manages `XMLHttpRequest` (XHR) requests and responses. The management code is necessary to control the busy indicators and preserve execution order. I didn't include any code in Chapter 4, but this section presents the execution order part of the implementation.

The XHR Injection technique does not preserve execution order in any browser, as shown in Table 5-1. The `EFWS.Script` module wraps this technique with code that queues up the XHR responses and makes sure they are executed in order. The implementation requires fewer than 100 lines of code:

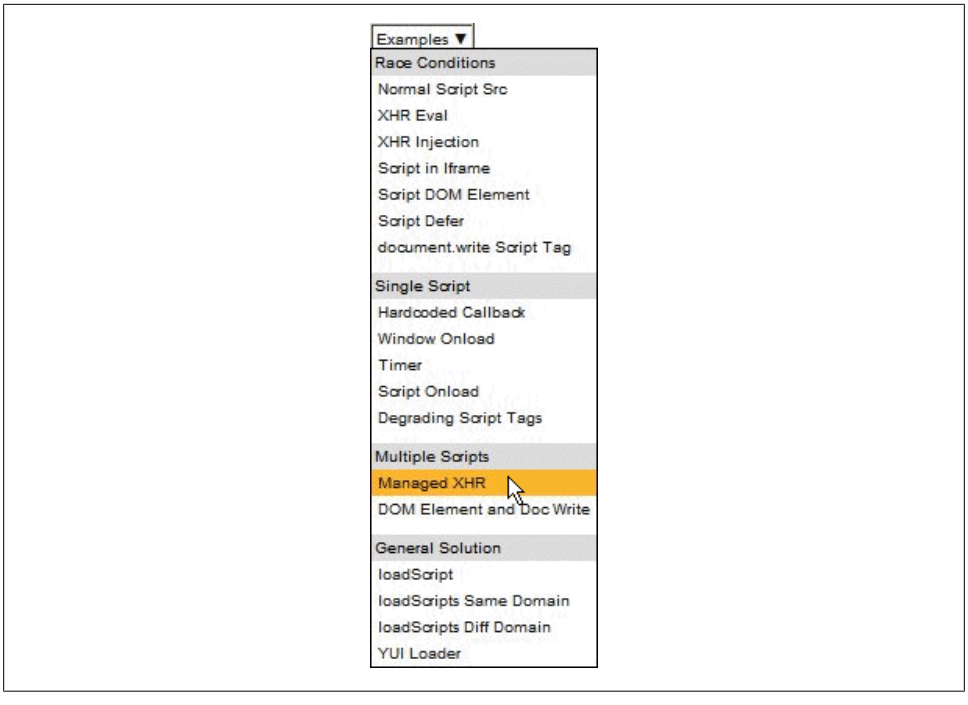


Figure 5-4. menutier.js example

```
<script type="text/javascript">
EFWS.Script = {
    queuedScripts: new Array(),

    loadScriptXhrInjection: function(url, onload, bOrder) {
        var iQ = EFWS.Script.queuedScripts.length;
        if ( bOrder ) {
            var qScript = {response: null, onload: onload, done: false};
            EFWS.Script.queuedScripts[iQ] = qScript;
        }

        var xhrObj = EFWS.Script.getXHRObject();
        xhrObj.onreadystatechange = function() {
            if ( xhrObj.readyState == 4 ) {
                if ( bOrder ) {
                    EFWS.Script.queuedScripts[iQ].response =
                        xhrObj.responseText;
                    EFWS.Script.injectScripts();
                }
                else {
                    eval(xhrObj.responseText);
                    if ( onload ) {
                        onload();
                    }
                }
            }
```

```
        };
        xhrObj.open('GET', url, true);
        xhrObj.send('');
    },

    injectScripts: function() {
        var len = EFWS.Script.queuedScripts.length;
        for ( var i = 0; i < len; i++ ) {
            var qScript = EFWS.Script.queuedScripts[i];
            if ( ! qScript.done ) {
                if ( ! qScript.response ) {
                    // STOP! need to wait for this response
                    break;
                }
                else {
                    eval(qScript.response);
                    if ( qScript.onload ) {
                        qScript.onload();
                    }
                    qScript.done = true;
                }
            }
        }
    },

    getXHRObject: function() {
        var xhrObj = false;
        try {
            xhrObj = new XMLHttpRequest();
        }
        catch(e){
            var aTypes = ["Msxml2.XMLHTTP.6.0",
                          "Msxml2.XMLHTTP.3.0",
                          "Msxml2.XMLHTTP",
                          "Microsoft.XMLHTTP"];
            var len = aTypes.length;
            for ( var i=0; i < len; i++ ) {
                try {
                    xhrObj = new ActiveXObject(aTypes[i]);
                }
                catch(e) {
                    continue;
                }
                break;
            }
        }
        finally {
            return xhrObj;
        }
    }
};
</script>
```

The `queuedScripts` array holds scripts that are queued for execution. Each queued script is an object with three properties:

`Response`
: The XHR response (a JavaScript string)

`Onload`
: A function to invoke once the script is loaded (optional)

`bOrder`
: `True` if this script must be executed in order with regard to other scripts (default is `false`)

Developers call `EFWS.Script.loadScriptXhrInjection`, passing in the URL of the external script to load, an `onload` function, and a Boolean indicating whether execution order should be preserved. If order doesn't matter, the XHR response is injected into the page as soon as it returns. When order does matter, the XHR response is added to the `queuedScripts` array and `EFWS.Script.injectScripts` is called. This function iterates over the queued scripts and injects any unexecuted responses, provided that all its dependencies have already been loaded. The Managed XHR example demonstrates this code.

Managed XHR
: *http://stevesouders.com/efws/managed-xhr.php*

The modified inline script follows. The first few lines are similar to the earlier examples; arrays of menu items and URLs are created. The `init` function makes the call to `EFWS.Menu.createTieredMenu`. The last two lines are where Managed XHR is used:

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus =
    [
     ["Race Conditions", aRaceConditions],
     ["Workarounds", aWorkarounds],
     ["Multiple Scripts", aMultipleScripts],
     ["General Solution", aLoadScripts]
    ];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

EFWS.Script.loadScriptXhrInjection("menu.js", null, true);
EFWS.Script.loadScriptXhrInjection("menutier.js", init, true);
</script>
```

The first call to `EFWS.Script.loadScriptXhrInjection` loads *menu.js* with execution order preserved. The second call causes *menutier.js* to be downloaded. It also is specified to be loaded in order, and `init` is passed in as this script's `onload` function.

The HTTP waterfall chart for this example, Figure 5-5, shows a short request for the HTML document, followed by requests for the three resources in the page: *menu.js* (two-second response), *menutier.js* (one-second response), and the image (one-second response). All of the resources in the page load in parallel and execution order is preserved (no undefined symbol errors occur).

Figure 5-5. Managed XHR HTTP waterfall chart

Managed XHR solves the problem across all the major browsers. However, this technique won't work if the external scripts are hosted on a different domain than the main page, due to the same-origin policy for `XMLHttpRequest`.* The DOM Element and Doc Write technique is the solution to use when your scripts are on a different domain than the main page.

DOM Element and Doc Write

Managed XHR works well for loading external and inline scripts in the order specified, while also loading scripts without blocking other resources in the page. Unfortunately, it can be used only for scripts on the same domain as the main page. It's not unusual for external scripts to reside on a domain that differs from the main page, especially when hosting your scripts on a Content Delivery Network (CDN) or using a third-party JavaScript library. The DOM Element and Doc Write example creates this situation by requesting *menu.js* and *menutier.js* from *http://souders.org*, while the main page still resides on *http://stevesouders.com*.

DOM Element and Doc Write
http://stevesouders.com/efws/dom-and-docwrite.php

Three asynchronous loading techniques can be used for scripts on a different domain: Script DOM Element, Script Defer, and `document.write` Script Tag. (See Chapter 4 for a description of each technique.) These techniques behave differently depending on the browser. Table 5-2 shows the results of measuring three traits, listed in priority order:

* *http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_XMLHttpRequest*

- Is the execution order of scripts preserved?
- Do scripts load in parallel with other scripts?
- Do scripts load in parallel with other resources (images, stylesheets, etc.)?

Table 5-2. Loading scripts asynchronously while preserving order

Technique	Preserve order	Scripts load in parallel	Other resources load in parallel
Script DOM Element	FF, Op	FF, Op, IE, Saf, Chr	IE, FF, Saf, Chr
Script Defer	IE, Saf, Chr, FF, Op	IE	IE
`document.write` Script Tag	IE, Saf, Chr, FF, Op	IE, Op	

Abbreviations are as follows: (Chr) Chrome 1.0.154 and 2.0.156; (FF) Firefox 2.0, 3.0, and 3.1; (IE) Internet Explorer 6, 7, and 8; (Op) Opera 9.63 and 10.00 alpha; (Saf) Safari 3.2.1 and 4.0 (developer preview).

Script DOM Element is the preferred technique for Firefox and Opera. In all other cases, `document.write` Script Tag is used. Script Defer is not used, even for Internet Explorer, because it can produce unexpected behavior when combined with DHTML techniques. I extended the `EFWS.Script` module to include these techniques:

```
EFWS.Script = {
    loadScriptDomElement: function(url, onload) {
        var domscript = document.createElement('script');
        domscript.src = url;
        if ( onload ) {
            domscript.onloadDone = false;
            domscript.onload = onload;
            domscript.onreadystatechange = function() {
                if ( "loaded" === domscript.readyState &&
                     domscript.onloadDone ) {
                    domscript.onloadDone = true;
                    domscript.onload();
                }
            }
        }
        document.getElementsByTagName('head')[0].appendChild(domscript);
    },

    loadScriptDocWrite: function(url, onload) {
        document.write('<scr' + 'ipt src="' + url +
                       '" type="text/javascript"></scr' + 'ipt>');
        if ( onload ) {
            EFWS.addHandler(window, "load", onload);
        }
    },

    queuedScripts: new Array(),
    loadScriptXhrInjection: function(url, onload, bOrder) { ... },
    injectScripts: function() { ... },
```

```
    getXHRObject: function() { ... }
};

EFWS.addHandler = function(elem, type, func) {
    if ( elem.addEventListener ) {
        elem.addEventListener(type, func, false);
    }
    else if ( elem.attachEvent ) {
        elem.attachEvent("on" + type, func);
    }
};
```

In addition to loading scripts without blocking and preserving execution order, we also want to couple the external script with inlined code. After all, that's the point of this chapter. In `EFWS.Script.loadScriptDomElement`, this is done by adding `onload` and `onreadystatechange` callbacks to the external script, as described in "Technique 4: Script Onload" on page 49. Although it's less preferred, we use Window Onload as the coupling technique in `EFWS.Script.loadScriptDocWrite` because the other techniques aren't possible when using `document.write` to insert the external script.

The inlined code in this section's example uses these new techniques, with special casing based on the browser:

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

if ( -1 != navigator.userAgent.indexOf('Firefox') ||
     -1 != navigator.userAgent.indexOf('Opera') ) {
    EFWS.Script.loadScriptDomElement("http://souders.org/efws/menu.js");
    EFWS.Script.loadScriptDomElement("http://souders.org/efws/menutier.js", init);
}
else {
    EFWS.Script.loadScriptDocWrite("http://souders.org/efws/menu.js");
    EFWS.Script.loadScriptDocWrite("http://souders.org/efws/menutier.js", init);
}
</script>
```

Combining Script DOM Element and `document.write` Script Tag accomplishes our primary goals. Execution order of external scripts is preserved across all browsers. The inlined code is successfully coupled with the external script on which it depends. Asynchronous loading is achieved to different degrees across browsers:

- Firefox loads all resources in parallel.
- Internet Explorer and Opera load scripts in parallel with other scripts, but other resources (images, stylesheets, etc.) are blocked.

- The results are mixed in Safari and Chrome. Safari 3.2 and Chrome 1.0 don't load any resources in parallel. However, using these same techniques in Safari 4 and Chrome 2.0 results in all resources loading in parallel.

As shown in this section, there's no easy cross-browser solution to loading multiple scripts asynchronously while preserving execution order. One option to consider is combining all your scripts into a single script. This is one of the best practices from *High Performance Web Sites* ("Rule 1: Make Fewer HTTP Requests") because it reduces download time. The additional benefit is that there's a more robust solution for loading single scripts asynchronously while coupling with inline code.

General Solution

This chapter presents many techniques, along with web page examples and code samples. It's valuable to understand the trade-offs, but what's needed is a general solution for loading scripts asynchronously while preserving execution order and coupling with inlined code. Building on top of the `EFWS.Script` functionality built so far, I add two new functions that hide all the details: `EFWS.Script.loadScript` for loading a single script, and `EFWS.Script.loadScripts` for loading multiple scripts.

Single Script

The best technique for loading a single script asynchronously is Script DOM Element. It works across all browsers and is lightweight. The Script Onload pattern is the best choice for coupling inlined code with an external script. `EFWS.Script.loadScriptDomElement` implements both of these techniques, so the general solution for single scripts is just a wrapper for this function:

```
EFWS.Script = {
    loadScript: function(url, onload) {
        EFWS.Script.loadScriptDomElement(url, onload);
    },

    loadScriptDomElement: function(url, onload) { ... },
    loadScriptDocWrite: function(url, onload) { ... },
    queuedScripts: new Array(),
    loadScriptXhrInjection: function(url, onload, bOrder) { ... },
    injectScripts: function() { ... },
    getXHRObject: function() { ... }
};
```

This greatly simplifies the *menu.js* example. The inlined code becomes just a few lines—the array of menu items, the `init` function, and a call to `EFWS.Script.loadScript`:

```
<script type="text/javascript">
var aExamples = [['couple-normal.php', 'Normal Script Src'],...];

function init() {
    EFWS.Menu.createMenu('examplesbtn', aExamples);
```

```
}

EFWS.Script.loadScript("menu.js", init);
</script>
```

The loadScript example demonstrates this code.

loadScript
: *http://stevesouders.com/efws/loadscript.php*

Multiple Scripts

The name of the general solution function for multiple scripts is `EFWS.Script.loadScripts`. "Multiple External Scripts" on page 52 discusses the techniques used in this situation. Managed XHR is the preferred solution when scripts are from the same domain as the main page. Asynchronously loading scripts from a different domain while preserving execution order is trickier because of fewer options and inconsistencies across browsers. The approach that's described in "DOM Element and Doc Write" on page 56 uses the Script DOM Element and `document.write` Script Tag techniques, depending on the browser. To show both cases, there are two examples that use this new `EFWS.Script.loadScripts` function.

loadScripts Same Domain
: *http://stevesouders.com/efws/loadscripts-same.php*

loadScripts Different Domain
: *http://stevesouders.com/efws/loadscripts-diff.php*

The code for `EFWS.Script.loadScripts` follows. `EFWS.Script.loadScripts` accepts an array of script URLs and a function to call after the last external script is done executing. `EFWS.Script.loadScripts` starts off by iterating over the script URLs to determine whether they're all from the same domain as the main page. This is done because a single technique must be used if all the external scripts are to be loaded in order. If they are from the same domain, `EFWS.Script.loadScriptXhrInjection` is chosen as the script loading function. If the scripts are served from a different domain, then `EFWS.Script.loadScriptDomElement` is used for Firefox and Opera, and `EFWS.Script.loadScriptDocWrite` is used for all others. (See "Multiple External Scripts" on page 52 for an explanation of why these alternatives are chosen.)

```
EFWS.Script = {
    loadScripts: function(aUrls, onload) {
        // first pass: see if any of the scripts are on a different domain
        var nUrls = aUrls.length;
        var bDifferent = false;
        for ( var i = 0; i < nUrls; i++ ) {
            if ( EFWS.Script.differentDomain(aUrls[i]) ) {
                bDifferent = true;
                break;
            }
        }
```

```
            // pick the best loading function
            var loadFunc = EFWS.Script.loadScriptXhrInjection;
            if ( bDifferent ) {
                if ( -1 != navigator.userAgent.indexOf('Firefox') ||
                     -1 != navigator.userAgent.indexOf('Opera') ) {
                    loadFunc = EFWS.Script.loadScriptDomElement;
                }
                else {
                    loadFunc = EFWS.Script.loadScriptDocWrite;
                }
            }

            // second pass: load the scripts
            for ( var i = 0; i < nUrls; i++ ) {
                loadFunc(aUrls[i], ( i+1 == nUrls ? onload : null ), true);
            }
        },

        differentDomain: function(url) {
            if ( 0 === url.indexOf('http://') || 0 === url.indexOf('https://') ) {
                var mainDomain = document.location.protocol + "://" +
                                 document.location.host + "/";
                return ( 0 !== url.indexOf(mainDomain) );
            }

            return false;
        },

        loadScript: function(url, onload) { ... },
        loadScriptDomElement: function(url, onload) { ... },
        loadScriptDocWrite: function(url, onload) { ... },
        queuedScripts: new Array(),
        loadScriptXhrInjection: function(url, onload, bOrder) { ... },
        injectScripts: function() { ... },
        getXHRObject: function() { ... }
    };
```

Once the appropriate loading function is determined, a second pass through the array of script URLs is performed to load each script. It's important to note that `true` is passed as the third argument to the script loading function. This is critical when `EFWS.Script.loadScriptXhrInjection` is the loading function so that responses are executed in the specified order. This parameter is ignored by `EFWS.Script.loadScriptDomElement` and `EFWS.Script.loadScriptDocWrite`, because those techniques preserve script execution order by default—that's why they were chosen.

The loadScripts Same Domain example uses *menu.js* and *menutier.js*, but now the script loading code is one line:

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
```

```
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

EFWS.Script.loadScripts( ["menu.js", "menutier.js"], init);
</script>
```

The loadScripts Different Domain example uses *menu.js* and *menutier.js* served from *http://souders.org*. The script loading code is still just one (wrapped) line of code:

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

EFWS.Script.loadScripts( ["http://souders.org/efws/menu.js",
                          "http://souders.org/efws/menutier.js"], init);
</script>
```

In these examples, `EFWS.Script.loadScripts` successfully loads scripts asynchronously while preserving execution order. The asynchronous loading of other resources (the image in this case) varies by browser, as documented earlier in Table 5-2. Firefox 2 and 3, Safari 4, and Chrome 2 load the image in parallel with the scripts, resulting in a waterfall chart such as that shown in Figure 5-6. The image is blocked from downloading in Internet Explorer 6 through 8, Opera, Safari 3, and Chrome 1, resulting in a longer load time as shown in Figure 5-7. Although the asynchronous loading of the image has mixed results, the scripts are loaded in parallel in all browsers except Safari 3 and Chrome 1.

Figure 5-6. loadScripts Different Domain HTTP waterfall chart, Firefox 3

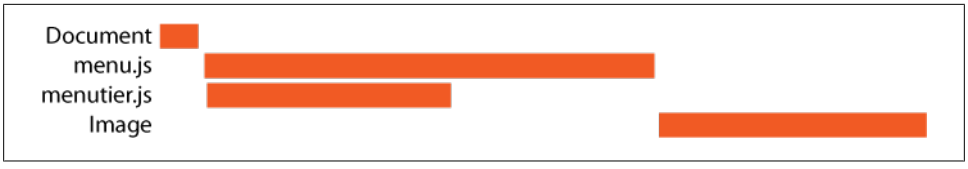


Figure 5-7. loadScripts Different Domain HTTP waterfall chart, Internet Explorer 7

Asynchronicity in the Real World

In this section, I review how some popular JavaScript frameworks do script loading.

Google Analytics and Dojo

I've mentioned Google Analytics in this chapter. It is a service from Google that web developers can use to gather web site metrics. The functionality is wrapped inside *http://www.google-analytics.com/ga.js*. The Google Analytics Help Center recommends adding this external script to a web site using `document.write`:†

```
<script type="text/javascript">
var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." :
"http://www.");
document.write(unescape("%3Cscript src='" + gaJsHost + "google-
analytics.com/ga.js' type='text/javascript'%3E%3C/script%3E"));
</script>
<script type="text/javascript">
var pageTracker = _gat._getTracker("UA-xxxxxx-x");
pageTracker._trackPageview();
</script>
```

This is a great example to analyze in the context of this chapter. The external script is a good candidate for asynchronous loading since it isn't used for rendering the visible page. The inline script depends on the external script, so execution order must be preserved and they must be coupled together.

The `document.write` Script Tag approach in the Google Analytics recommendation has some benefits. The URL is dynamically modified to load over HTTPS if appropriate. The execution order of the external script and inlined code is preserved across all browsers.

A drawback of the `document.write` Script Tag technique is that it blocks other resources from being downloaded. The `dojox.analytics.Urchin` module addresses this issue, as described in the first line from The Dojo Foundation's documentation page:‡

† *http://www.google.com/support/analytics/bin/answer.py?hl=en&answer=55488*

‡ *http://docs.dojocampus.org/dojox/analytics/Urchin*

> This class is used to delay loading of the popular Google Analytics Tracker, formerly known as Urchin. The synchronous nature of <script> tags causes page rendering to stall until loading of remote files has completed, and this module alleviates that.

`dojox.analytics.Urchin` is part of the Dojo JavaScript toolkit. As the documentation points out, Urchin is the former name for the Google Analytics module. This explains why the Dojo module is named *Urchin.js*. The key functions in this module are `_loadGA`, `_checkGA`, and `_gotGA`:§

```
_loadGA: function(){
    // summary: load the ga.js file and begin initialization process
    var gaHost = ("https:" == document.location.protocol) ? "https://ssl." :
"http://www.";
    dojo.create('script', {
        src: gaHost + "google-analytics.com/ga.js"
    }, dojo.doc.getElementsByTagName("head")[0]);
    setTimeout(dojo.hitch(this, "_checkGA"), this.loadInterval);
},

_checkGA: function(){
    // summary: sniff the global _gat variable Google defines and either check
again
    //          or fire onLoad if ready.
    setTimeout(dojo.hitch(this, !window["_gat"] ? "_checkGA" : "_gotGA"),
this.loadInterval);
},

_gotGA: function(){
    // summary: initialize the tracker
    this.tracker = _gat._getTracker(this.acct);
    this.tracker._initData();
    this.GAonLoad.apply(this, arguments);
},
```

The `_loadGA` function uses the Script DOM Element asynchronous loading technique. It calls `dojo.create` to create a script element, setting its `src` to *http://www.google-analytics.com/ga.js* or *https://ssl.google-analytics.com/ga.js*, depending on the protocol of the main page. The script element is appended to the document's head.

Coupling *ga.js* with the inlined code is done with a timer. Every `loadInterval` (420 milliseconds), `_checkGA` is called to see whether `window["_gat"]` (the Google Analytics object) is defined. If so, `_gotGA` is called to instantiate the Google Analytics tracker. This coupling approach is similar to the Timer technique described in "Technique 3: Timer" on page 48.

Comparing this implementation to `EFWS.Script.loadScript`, we see that both use the Script DOM Element approach. Using this technique allows the script to be downloaded without blocking other resources and works in all major browsers. The coupling technique is different, though. Instead of the Timer technique,

§ *http://bugs.dojotoolkit.org/browser/dojox/trunk/analytics/Urchin.js*. Copyright (c) 2004–2008, The Dojo Foundation. All rights reserved. See *http://dojotoolkit.org/license* for details.

`EFWS.Script.loadScript` uses the Script Onload technique. The Timer technique has disadvantages:

- If the script fails to load, the timer will continue indefinitely.
- This approach requires more maintenance. If *ga.js* changes and no longer defines `_gat`, then `_checkGA` would have to be updated. The Script Onload approach doesn't rely on any of the symbols in *ga.js*.
- There can be a delay of up to 420 milliseconds between when *ga.js* is done loading and when `_gotGA` is called. That's enough time for the user to leave the page before the tracker can do its work. The Script Onload approach calls the inlined code as soon as the external script is loaded.

For these reasons, Script Onload is the coupling technique chosen in `EFWS.Script.loadScript`.

YUI Loader Utility

Google Analytics is a good example for analyzing how to load a single script asynchronously while coupling it with inlined code. The YUI Loader Utility is the example I've chosen to examine how multiple scripts are loaded. This utility is part of the Yahoo! UI Library and is described as follows:‖

> The YUI Loader Utility is a client-side JavaScript component that allows you to load specific YUI components and their dependencies into your page via script. YUI Loader can operate as a holistic solution by loading all of your necessary YUI components, or it can be used to add one or more components to a page on which some YUI content already exists.

YUI Loader's objective is to provide anytime loading and dependency calculation. It improves page performance by pulling in only the modules that are necessary and combining those into a single HTTP request, thanks to combo-handling.# I converted the example that uses *menu.js* and *menutier.js* to use YUI Loader in order to see whether scripts get loaded in parallel.

YUI Loader
: *http://stevesouders.com/efws/yuiloader.php*

This example starts by loading the YUI Loader itself from *http://yui.yahooapis.com/2.6.0/build/yuiloader/yuiloader-min.js*. An instance of `YUILoader` is created and `addModule` is used to load *menu.js* and *menutier.js*. The `init` function is specified to be called after these scripts are successfully loaded. Everything is kicked off by calling `insert`:

```
<script type="text/javascript"
src="http://yui.yahooapis.com/2.6.0/build/yuiloader/yuiloader-min.js">
</script>
```

‖ *http://developer.yahoo.com/yui/yuiloader/*

http://yuiblog.com/blog/2008/10/17/loading-yui/

```
<script type="text/javascript">
var aRaceConditions = [['couple-normal.php', 'Normal Script Src'], ...];
var aWorkarounds = [['hardcoded-callback.php', 'Hardcoded Callback'], ...];
var aMultipleScripts = [['managed-xhr.php', 'Managed XHR'], ...];
var aLoadScripts = [['loadscript.php', 'loadScript'], ...];
var aSubmenus = [["Race Conditions", aRaceConditions], ...];

function init() {
  EFWS.Menu.createTieredMenu('examplesbtn', aSubmenus);
}

var loader = new YAHOO.util.YUILoader();
loader.addModule({ name: "menu", type: "js", fullpath: "menu.js"});
loader.addModule({ name: "menutier", type: "js", fullpath: "menutier.js"});
loader.require("menu");
loader.require("menutier");
loader.onSuccess = init;
loader.insert();
</script>
```

We can look at how YUI Loader is implemented in *http://yui.yahooapis.com/2.6.0/build/yuiloader/yuiloader.js* (the commented, unminified version of the code). The scripts are inserted by the `_node` function in a way similar to the Script DOM Element approach. The `_track` function uses the Script Onload coupling technique. YUI's implementation is extremely thorough, with special handling for browser edge cases.

The most important observation is that YUI Loader does not load the scripts in parallel, even though Script DOM Element is used. YUI Loader explicitly loads scripts sequentially, waiting for the first script to return before requesting the next one. This can be seen in the example's HTTP waterfall chart, shown in Figure 5-8. The scripts are the last two requests. Comparing this to Figures 5-6 and 5-7, we see that `EFWS.Script.loadScripts` loads the scripts in parallel, resulting in a faster page.

Figure 5-8. YUI Loader HTTP waterfall chart

The sequential loading behavior of YUI Loader causes the scripts to take longer to load than `EFWS.Script.loadScripts` in all browsers except Safari 3 and Chrome 1. To be fair, YUI Loader is capable of loading scripts anytime, even after the document has loaded. `EFWS.Script.loadScripts`, with its use of `document.write` in some browsers, can be used only while the document is loading.

For pages with external scripts in the main page, loading them asynchronously with `EFWS.Script.loadScripts` improves performance, and this benefit is more pronounced as the number of scripts increases. A simpler alternative is to concatenate the scripts together, as recommended in Rule 1 from *High Performance Web Sites*. But that's not always possible. Across the top 10 U.S. web sites, the average number of external scripts is 6.5 (see Table 11-1). Loading these scripts in parallel, while preserving execution order and coupling inlined code, is critical to making today's popular web sites faster for users.

CHAPTER 6

Positioning Inline Scripts

The previous three chapters focused on the impact of *external* scripts. This chapter focuses on *inline* scripts (JavaScript included in the HTML document directly). Even though inline scripts don't introduce additional HTTP requests, they can block resources in the page from being downloaded in parallel. They can also thwart progressive rendering. This chapter explains why the decisions of when and where to inline JavaScript have an impact on page performance.

Inline Scripts Block

Chapter 5 describes how external scripts block parallel downloads and rendering. It's not surprising that inline scripts have the same behavior for the same reasons (preserving execution order and `document.write` dependencies). The Inline Scripts Block example demonstrates this behavior.

Inline Scripts Block
http://stevesouders.com/cuzillion/?ex=10100&title=Inline+Scripts+Block

Figure 6-1 shows the HTTP requests issued for this page. In addition to the HTML document, there are two image requests, each configured to take one second. An inline script is inserted between these two images. The inline script is represented by a line in Figure 6-1. It does not generate an HTTP request, but the impact it has is observable.

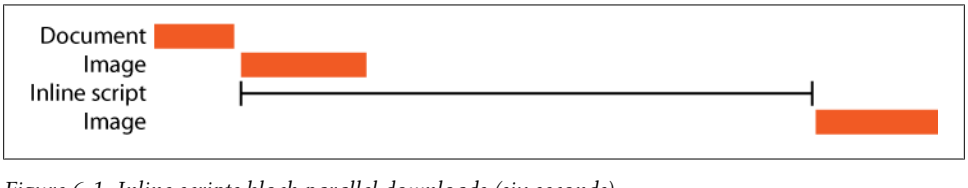


Figure 6-1. Inline scripts block parallel downloads (six seconds)

The inline script is configured to take five seconds to execute. This is what causes the four seconds of whitespace between the two image requests in Figure 6-1. The inline script starts executing in parallel with the first image request. After one second, the image response is received, but the inline script continues to execute for another four seconds. While the inline script is executing, all other downloads are blocked. It's not until the inline script finishes (five seconds into the page) that the second image finally starts to download, resulting in an overall load time of six seconds.

In addition to blocking parallel downloads, inline scripts block rendering. When the Inline Scripts Block page is loaded, nothing in the page is painted for at least five seconds. The best way to observe this is to first set the browser location to another page or `about:blank`, and then visit the Inline Scripts Block URL. Five seconds pass before anything is rendered. This is surprising because some plain text is in the HTML document (the "Cuzillion" header, the "Examples" and "Help" links, etc.) before the inline script, but the browser doesn't render this until the inline script has finished executing.

If your site uses inline scripts, it's important to understand how they block downloads and rendering, and to avoid this behavior if possible. Several workarounds are available:

- Move inline scripts to the bottom.
- Initiate the JavaScript execution using an asynchronous callback.
- Use the `SCRIPT DEFER` attribute.

Each technique is explained in the following sections.

Move Inline Scripts to the Bottom

Parallel downloading and progressive rendering are achieved by moving inline scripts below all the resources in the page.* The benefit of moving inline scripts to the bottom is demonstrated in the following example.

Move Inline Scripts to the Bottom
http://stevesouders.com/efws/inline-scripts-bottom.php

Figure 6-2 shows the two image requests downloading in parallel. The five-second inline script executes in parallel as well, resulting in an overall page load time of five seconds, one second faster than the baseline page. Although this technique avoids blocking downloads, rendering is still blocked. If your inline scripts don't take very long (fewer than 300 milliseconds) to execute, this technique is an easy way to speed up your pages. Inline scripts that take longer to execute should use one of the remaining two techniques.

* This is similar to the advice from *High Performance Web Sites*, "Rule 6: Put Scripts at the Bottom."

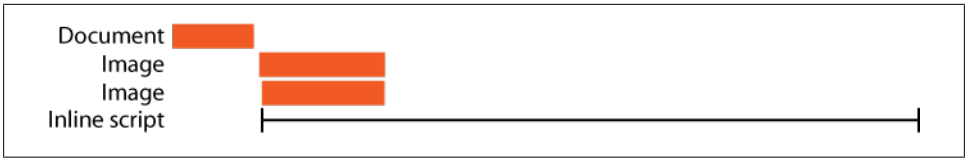


Figure 6-2. Inline scripts block parallel downloads (five seconds)

Initiate Execution Asynchronously

You can instruct a browser to execute an inline script asynchronously so that the browser has an opportunity to perform parallel downloads and progressive rendering. A simple asynchronous callback technique is to use `setTimeout`, as shown in the following example:

```
function longCode() {
    var tStart = Number(new Date());
    while( (tStart + 5000) > Number(new Date()) ) {};
}

setTimeout(longCode, 0);
```

The function `longCode` kicks off JavaScript that takes five seconds to execute. In our first attempt at using `setTimeout`, we might use a value of zero milliseconds for the delay, as in the following example.

Inline Scripts via setTimeout (0 milliseconds)
http://stevesouders.com/efws/inline-scripts-settimeout.php?d=0

The results are similar to the Move Inline Scripts to the Bottom technique: the images are downloaded in parallel and the page takes five seconds to load. But unlike the previous technique, using `setTimeout` has the added benefit of progressive rendering in Internet Explorer. Before the inline script starts executing, there is enough time for Internet Explorer to render the text at the top of the page ("Cuzillion," the "Examples" and "Help" links, etc.).

Although `setTimeout` with a delay of zero milliseconds allows progressive rendering in Internet Explorer, Firefox rendering is still blocked. We need to increase the number of milliseconds to 250 to achieve progressive rendering in Firefox, which we do in the next example.

Inline Scripts via setTimeout (250 milliseconds)
http://stevesouders.com/efws/inline-scripts-settimeout.php?d=250

The magic number 250 comes from the default value for `nglayout.initialpaint.delay`. This is the "number of milliseconds to wait before first displaying the page."† If `longCode` kicks off before 250 milliseconds, all rendering is blocked until it

† *http://kb.mozillazine.org/Nglayout.initialpaint.delay*

finishes executing. If, however, we wait 250 milliseconds before calling `longCode`, Firefox is able to render the text at the top of the page.

In both cases (zero milliseconds for Internet Explorer and 250 milliseconds for Firefox), only the text is rendered quickly. The images, even though they return after one second, are not painted until `longCode` finishes five seconds into the page. The paint events are queued up at the one-second mark, but the browser isn't able to act on those events while `longCode` executes. The browser is single-threaded, while JavaScript executes all paint events are blocked.‡ We get around this in the next example by increasing the number of `setTimeout` milliseconds to a value slightly longer than the one-second download time of the images—for example, 1,500 milliseconds.

Inline Scripts via setTimeout (1,500 milliseconds)
http://stevesouders.com/efws/inline-scripts-settimeout.php?d=1500

Now the images are painted as soon as they download. Because it takes only one second to render everything in the page, the `onload` event fires after one second, as opposed to five seconds. One downside of using a 1,500-millisecond delay is that `longCode` doesn't finish executing until 6,500 milliseconds into the page (1,500-millisecond delay plus 5,000-millisecond execute time). If we want to asynchronously kick off `longCode` without blocking the browser from rendering the page, a better practice is to launch the code using the `onload` event:

```
function longCode() {
    var tStart = Number(new Date());
    while( (tStart + 5000) > Number(new Date()) ) {};
}

window.onload = longCode;
```

As shown in the following example, using the `onload` event lets the text and images on the page render as soon as they are available, and executes the inline script as early in the page as possible without blocking downloads and rendering.

Inline Scripts via onload
http://stevesouders.com/efws/inline-scripts-onload.php

If your inline scripts are short, using `setTimeout` with a delay of zero milliseconds is a good compromise between fast rendering and fast JavaScript execution. If your scripts are long, using `onload` is a better choice. The best solution is to yield every 300 milliseconds or so using `setTimeout`, but this can necessitate a significant redesign of your code to make it reentrant. See "Yielding Using Timers" on page 103 for an in-depth discussion of this technique.

‡ See Chapter 2 for more discussion of the impact of JavaScript on browser responsiveness.

Use Script Defer

The `SCRIPT DEFER` attribute for inline scripts is supported only in Internet Explorer and Firefox 3.1. Typically, people use it in conjunction with downloading external scripts, as described in Chapter 4. But the `DEFER` attribute is also applicable to inline scripts, where it allows the browser to continue parsing and rendering the page and postpone execution of the inline script. We can use Cuzillion to create an example of inline scripts that use the `DEFER` attribute.

Inline Scripts and Defer
http://stevesouders.com/cuzillion/?ex=10101&title=Inline+Scripts+and+Defer

Using `DEFER` in browsers that support it allows both images to be downloaded in parallel, resulting in an overall page load time of five seconds (faster than the six-second baseline). However, nothing is rendered in the page until the five-second script completes. `DEFER` is an easy workaround to enable parallel downloads, but it works only on inline scripts in Internet Explorer and Firefox 3.1, and still blocks progressive rendering. Using `setTimeout` is a better alternative.

Preserving CSS and JavaScript Order

The typical way to load external scripts is with the `SCRIPT SRC` attribute:

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

When scripts are loaded this way, they block parallel downloads, as described in Chapter 4. The main reason browsers download only one script at a time is to ensure proper execution order. Executing *B.js* before *A.js* might result in unexpected behavior or undefined symbols due to code dependencies.

Preserving the order of JavaScript is critical, and this is true for CSS as well. Given the cascading nature of styles, loading them in different orders may yield undesired results. To provide consistent behavior, browsers ensure that CSS is applied in the order specified. The Stylesheets in Order example confirms that stylesheets are applied in the order specified, regardless of the order in which the HTTP responses are received.

Stylesheets in Order
http://stevesouders.com/efws/stylesheets-order.php

This example has two stylesheets that define a rule with the same name. The first stylesheet is programmed to take longer to download, as shown in Figure 6-3. The first stylesheet specifies a gray background whereas the second stylesheet specifies an orange background. The color that wins out is orange, which means the second stylesheet was applied last even though it finished downloading first. This shows that browsers apply stylesheets in the order in which they are listed in the page, regardless of the order in which they are downloaded.

Figure 6-3. Stylesheets applied in order in Internet Explorer

The application of CSS is preserved across stylesheets and inline styles as well. In the CSS in Order example, the same long stylesheet from Figure 6-3 (with a gray background) is followed by an inline style (with an orange background). Again, the browser waits for the long stylesheet to download and applies it before the inline style to ensure that CSS is applied in the order specified in the page.

CSS in Order
http://stevesouders.com/efws/css-order.php

It's useful to know that browsers make sure to apply CSS in the order specified in the page. But what does this have to do with inline scripts? The next section pulls it all together.

Danger: Stylesheet Followed by Inline Script

In the previous section, we confirmed that browsers apply CSS (stylesheets as well as inline styles) in the order in which they appear in the HTML document. Earlier in this chapter, we verified that inline scripts block other browser activity (downloads and rendering). These insights are fairly well known in the web development community. What is less well known is that browsers also apply CSS and JavaScript sequentially, and that this behavior can significantly delay downloaded resources when a stylesheet is followed by an inline script. This sequence causes subsequent resources to be blocked until the stylesheet is downloaded and the inline script is executed. The following sections explain why this problem occurs.

Inline Scripts Aren't Blocked by Most Downloads

Inline scripts can execute while images and iframes are being downloaded, as shown in this example.

Inline Scripts After Image and Iframe
http://stevesouders.com/cuzillion/?ex=10102&title=Inline+Scripts+After

Figure 6-4 contains the HTTP profile for the Inline Scripts After Image and Iframe example. This shows three resources that each take two seconds to download: an image, an iframe, and another image. Between each of these is an inline script that takes one second to execute. The key events in the page load timeline are explained in the list that follows.

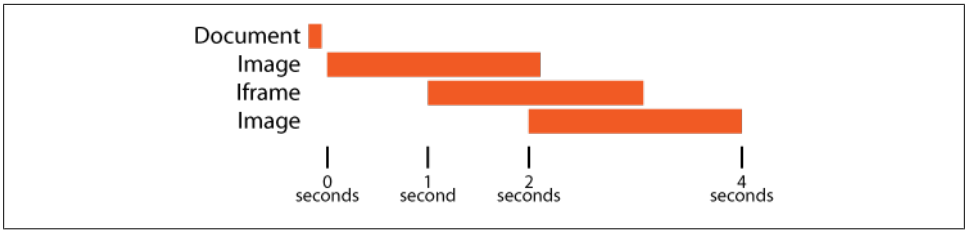


Figure 6-4. Inline scripts after an image and an iframe (four seconds)

0 seconds
: The first image starts downloading. The first inline script starts executing in parallel with the image download.

1 second
: The first inline script finishes. This opens the door for the iframe to start downloading and for the second inline script to start executing. The second inline script executes while the iframe is being downloaded.

2 seconds
: The second inline script finishes executing, allowing the final image to start downloading.

4 seconds
: The final image finishes downloading.

Because inline scripts execute while images and iframes are being downloaded, the overall page loads in just four seconds. Their interaction with stylesheets, however, blocks parallel downloads, as explained in the next section.

Inline Scripts Are Blocked by Stylesheets

The interaction between stylesheets and inline scripts is very different than with other resources. This is because browsers preserve the order in which CSS and JavaScript are parsed, as shown in this example.

Inline Scripts After Stylesheet
: *http://stevesouders.com/cuzillion/?ex=10103&title=Inline+Scripts+after*

The Inline Scripts After Stylesheet example is like the previous example, but the first image and iframe are replaced with stylesheets. As before, all of the resources take two seconds to download, and the inline scripts each take one second to execute. Figure 6-5 shows the HTTP profile. The overall load time is eight seconds, as compared to four seconds for the previous example! The page load timeline reveals why this page takes twice as long to load.

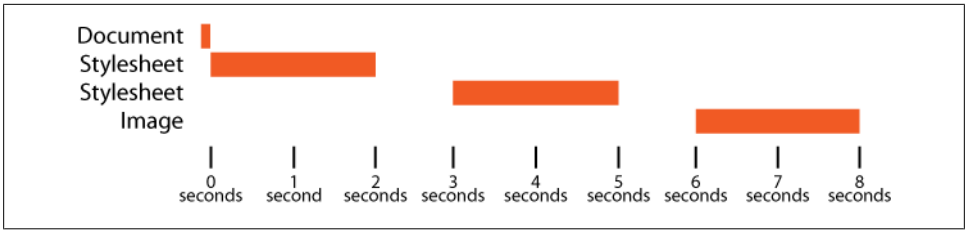


Figure 6-5. Inline scripts after stylesheets (eight seconds)

0 seconds
: The first stylesheet starts downloading. The first inline script is blocked from executing until the stylesheet is downloaded and parsed.

2 seconds
: The first stylesheet finishes downloading. The first inline script starts executing.

3 seconds
: The first inline script finishes executing. The second stylesheet starts downloading.

5 seconds
: The second stylesheet finishes downloading. The second inline script starts executing.

6 seconds
: The second inline script finishes executing. The image starts downloading.

8 seconds
: The image finishes downloading.

The way in which browsers process CSS and JavaScript sequentially causes this example to take twice as long as the previous one. This example shows that when confronted with a stylesheet followed by an inline script, browsers wait until the stylesheet is fully downloaded before starting to execute the inline script. Why is this? It's possible that the inline script contains code that depends on the styles applied from the stylesheet. I have seen and written JavaScript that does this. Further evidence that such JavaScript exists in the real world is the addition of `getElementsByClassName` in HTML 5.[§] Browsers download the stylesheet and execute the inline script sequentially in order to guarantee reproducible results.

This example also confirms that inline scripts block the download of any resources that follow them. Although resources typically download in parallel with stylesheets, the combination of these two constraints produces the key insight of this chapter: *stylesheets followed by an inline script block any subsequent resources from downloading*.

§ *http://dev.w3.org/html5/spec/Overview.html#dom-getelementsbyclassname*

This Does Happen

The previous examples illustrate the blocking that occurs when a stylesheet is followed by an inline script, but they seem, at least to me, a bit contrived. Fortunately (or unfortunately), because this behavior has not been widely examined or publicized, it's easy to find examples of this problem in the real world. Among the 10 top U.S. sites, four (eBay, MSN, MySpace, and Wikipedia) have a stylesheet followed by an inline script. This causes resources after the stylesheet to be downloaded later than necessary, resulting in a slower page.

Figure 6-6 shows part of eBay's HTTP profile, where the sequence of a stylesheet followed by an inline script causes two scripts to be blocked from downloading until the stylesheet is finished downloading. The arrow shows where the downloading could have started if not for this problem.

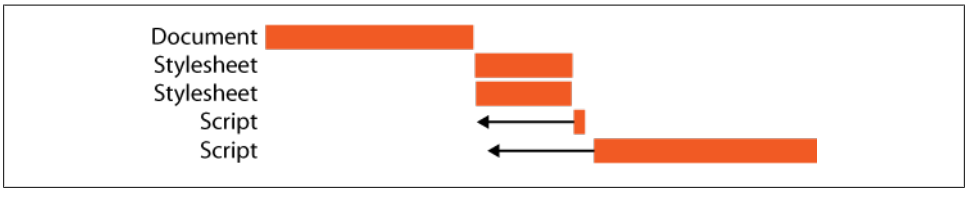


Figure 6-6. eBay stylesheet followed by inline script

Similarly, as shown in Figure 6-7, MSN has fewer parallel downloads than desired because an image is blocked by a stylesheet. In Figure 6-8, we see a MySpace script downloaded later because it's blocked by an inline script that occurs after the five stylesheets.

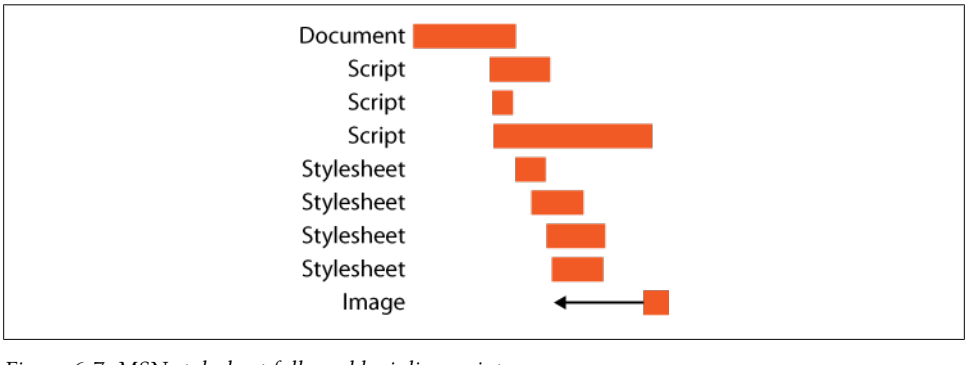


Figure 6-7. MSN stylesheet followed by inline script

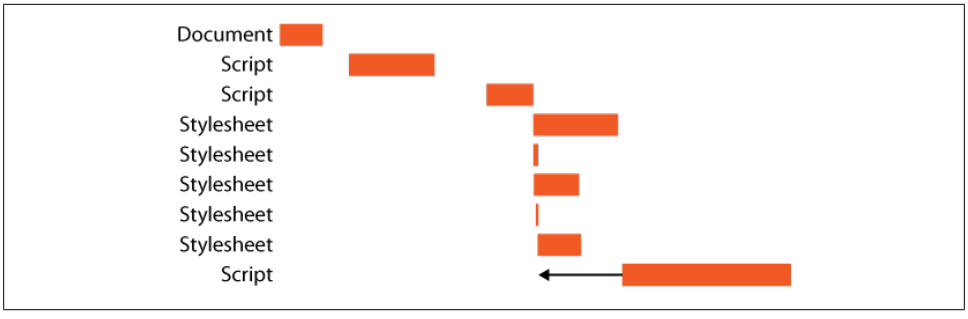


Figure 6-8. MySpace stylesheet followed by inline script

Wikipedia is shown in Figure 6-9. The script at the end of the HTTP profile is blocked from downloading because of a preceding stylesheet followed by an inline script. As a side note, this HTTP waterfall chart was produced using Internet Explorer 7, which supports two connections per hostname. And yet, this chart shows four connections working in parallel. That's because Wikipedia downgrades its traffic to HTTP/1.0, which increases the number of connections to four per hostname. (See "Downgrading to HTTP/1.0" on page 165.) However, the benefit of increased parallel downloads is forfeited when the script is downloaded due to the preceding stylesheet being followed by an inline script.

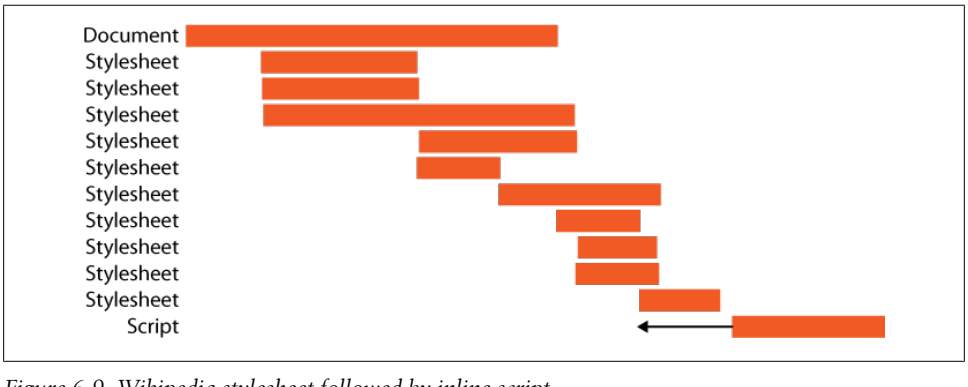


Figure 6-9. Wikipedia stylesheet followed by inline script

All of these problems are easily fixed. *The solution is to place inline scripts so that they do not appear between a stylesheet and any other resource*. The inline script should be placed either above the stylesheet or below the other resource. If the other resource is a script, there might be a code dependency between the inline script and the external script. For that reason, my general recommendation is to move the inline script above the stylesheet. This will avoid any code dependency issues. If you're certain there are no code dependencies, moving the inline script below the visible resources allows these resources to load sooner, resulting in better progressive rendering.

CHAPTER 7

Writing Efficient JavaScript

Nicholas C. Zakas

Today's web applications are powered by a large amount of JavaScript code. Whereas early web sites used JavaScript to perform simple tasks, the language is now used to run the entire user interface in many places. The result can be thousands of lines of JavaScript code to execute every time a user interaction takes place. Performance, therefore, is not just about how long it takes for the page to load, but also about how it responds as it's being used. The best way to ensure a fast, enjoyable user interface is to write JavaScript as efficiently as possible for all browsers.*

This chapter covers some of the hidden performance issues in JavaScript and how to address them. Some changes concern small code structure issues while others may require revisiting your algorithm. The important thing to remember is that there is no silver bullet when trying to improve performance; no one thing will work in 100% of the cases. Only when various techniques are combined can you realize the largest performance improvement.

Managing Scope

When JavaScript code is being executed, an *execution context* is created. The execution context (also sometimes called the *scope*) defines the environment in which code is to be executed. A global execution context is created upon page load, and additional execution contexts are created as functions are executed, ultimately creating an execution context stack where the topmost context is the active one.

Each execution context has a *scope chain* associated with it, which is used for identifier resolution. The scope chain contains one or more variable objects that define in-scope identifiers for the execution context. The global execution context has only one variable

* All of the research in this chapter is based on experiments run on Firefox versions 3.0 and 3.1 beta 2, Google Chrome 1.0, Internet Explorer versions 7 and 8 beta 2, Safari versions 3.0–3.2, and Opera version 9.62. When the version numbers aren't mentioned, all tested versions of the browser are relevant.

object in its scope chain, and this object defines all of the global variables and functions available in JavaScript. When a function is created (but not executed), its internal `[[Scope]]` property is assigned to contain the scope chain of the execution context in which it was created (internal properties cannot be accessed through JavaScript, so you cannot access this property directly). Later, when execution flows into a function, an activation object is created and initialized with values for `this`, `arguments`, named arguments, and any variables local to the function. The activation object appears first in the execution context's scope chain and is followed by the objects contained in the function's `[[Scope]]` property.

During code execution, identifiers such as variable and function names are resolved by searching the scope chain of the execution context. Identifier resolution begins at the front of the scope chain and proceeds toward the back. Consider the following code:

```
function add(num1, num2){
    return num1 + num2;
}

var result = add(5, 10);
```

When this code is executed, the `add` function has a `[[Scope]]` property that contains only the global variable object. As execution flows into the `add` function, a new execution context is created, and an activation object containing `this`, `arguments`, `num1`, and `num2` is placed into the scope chain. Figure 7-1 illustrates the behind-the-scenes object relationships that occur while the `add` function is being executed.

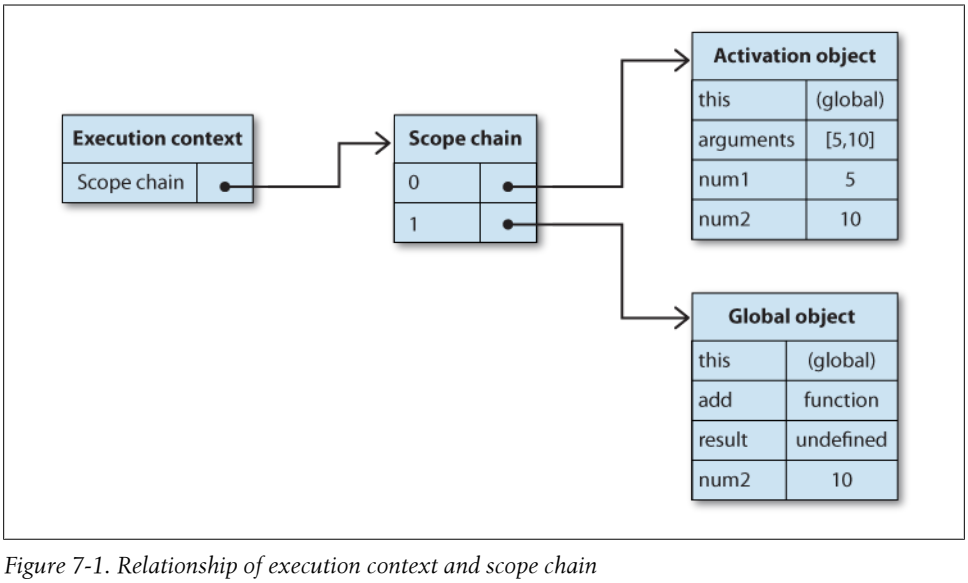


Figure 7-1. Relationship of execution context and scope chain

Inside the `add` function, the identifiers `num1` and `num2` need to be resolved when the function is executing. This resolution is performed by inspecting each object in the scope chain until the specific identifier is found. The search begins at the first object in the scope chain, which is the activation object containing the local variables for the function. If the identifier isn't found there, the next object in the scope chain is inspected for the identifier. When the identifier is found, the search stops. In the case of this example, the identifiers `num1` and `num2` exist in the local activation object and so the search never goes on to the global object.

Understanding scopes and scope chain management in JavaScript is important because identifier resolution performance is directly related to the number of objects to search in the scope chain. The farther up the scope chain an identifier exists, the longer the search goes on and the longer it takes to access that variable; if scopes aren't managed properly, they can negatively affect the execution time of your script.

Use Local Variables

Local variables are, by far, the fastest identifiers both to read from and write to in JavaScript. Because they exist in the activation object of the executing function, identifier resolution involves inspecting a single object in the scope chain. The amount of time necessary to read the value of a variable increases with each step along the scope chain, so the greater the identifier depth, the slower the access is going to be. This effect can be seen in every browser except Google Chrome using v8 and Safari 4+ using the Nitro JavaScript engine, both of which are so fast that the identifier depth has little effect on access speed.

To determine the exact performance impact of identifier depth, I ran an experiment involving 200,000 variable operations. I alternated between reads and writes, accessing the variables from different identifier depths. The page I used for this experiment is located at *http://www.nczonline.net/experiments/javascript/performance/identifier-depth/*.

Figure 7-2 illustrates the amount of time it takes to write to a variable based on scope chain depth, and Figure 7-3 illustrates the amount of time it takes to read from an identifier based on its scope chain depth (a depth of 1 signifies a local identifier).

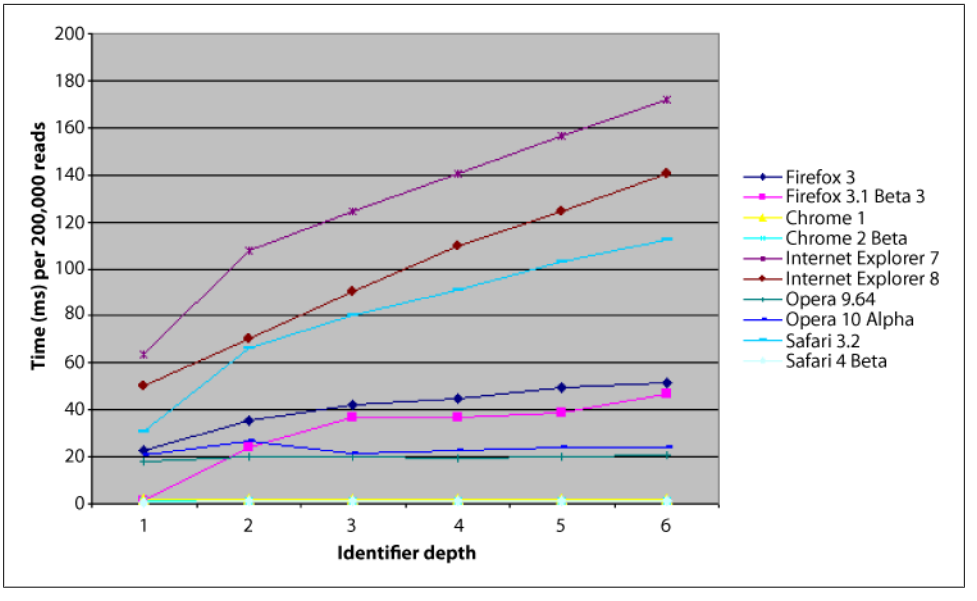


Figure 7-2. Variable read time compared to identifier depth

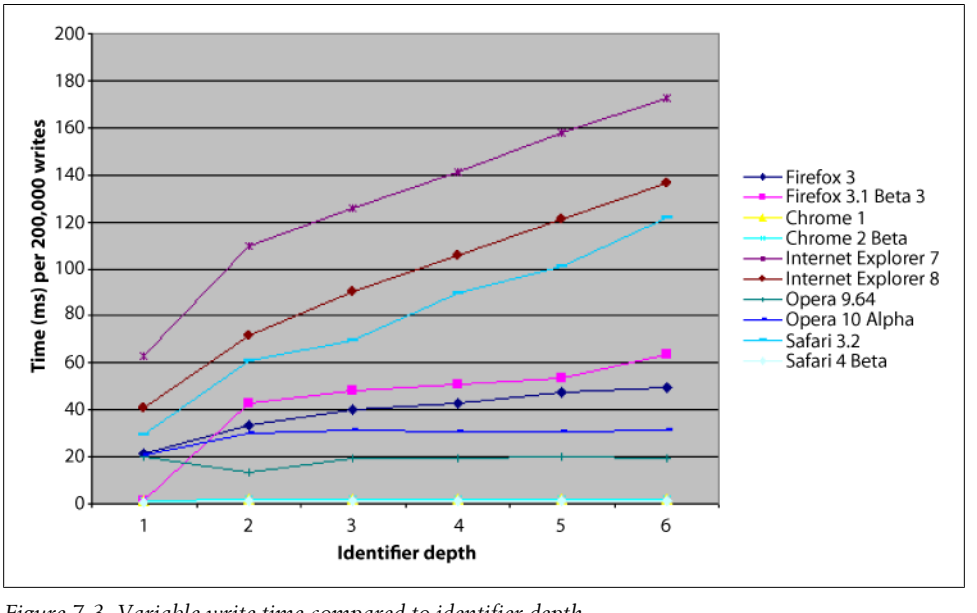


Figure 7-3. Variable write time compared to identifier depth

As these figures clearly indicate, identifiers are accessed significantly faster when they are higher in the scope chain. You can take advantage of this knowledge by using local variables whenever possible. A good rule of thumb is to store any out-of-scope variables in a local variable whenever it's used more than once within the function. For example:

```
function createChildFor(elementId){
    var element = document.getElementById(elementId),
        newElement = document.createElement("div");

    element.appendChild(newElement);
}
```

This function has two references to the global variable `document`. Since `document` is being used more than once, it should be stored in a local variable for faster reference, such as here:

```
function createChildFor(elementId){
    var doc = document,  //store in a local variable
        element = doc.getElementById(elementId),
        newElement = doc.createElement("div");

    element.appendChild(newElement);
}
```

The rewritten version of the function stores `document` in a local variable called `doc`. Since `doc` exists in the first part of the scope chain, it can be resolved faster than `document`. Keep in mind that the global variable object is always the last object in the scope chain, and so global identifier resolution is always the most expensive.

A very common mistake that leads to performance issues is to omit the `var` keyword when assigning a variable's value for the first time. Assignment to an undeclared variable automatically results in a global variable being created.

Scope Chain Augmentation

The scope chain for a given execution context typically remains unchanged during code execution. There are, however, two statements that temporarily augment the scope chain of an execution context. The first is the `with` statement, which is designed to allow easy access to object properties by making them appear as local variables. For example:

```
var person = {
    name: "Nicholas",
    age: 30
};

function displayInfo(){
    var count = 5;
    with(person){
        alert(name + " is " + age);
        alert("Count is " + count);
```

```
    }
}

displayInfo();
```

In this code, the `person` object is passed into a `with` block. This allows you to access the `name` and `age` properties as though they were locally defined. What actually happens, though, is that a new variable object is pushed to the front of the execution context's scope chain. This variable object contains all of the properties of the specified object (in this case, `person`) so that they can be accessed without using dot notation. Figure 7-4 shows how the scope chain for `displayInfo` is augmented while the `with` statement is being executed.

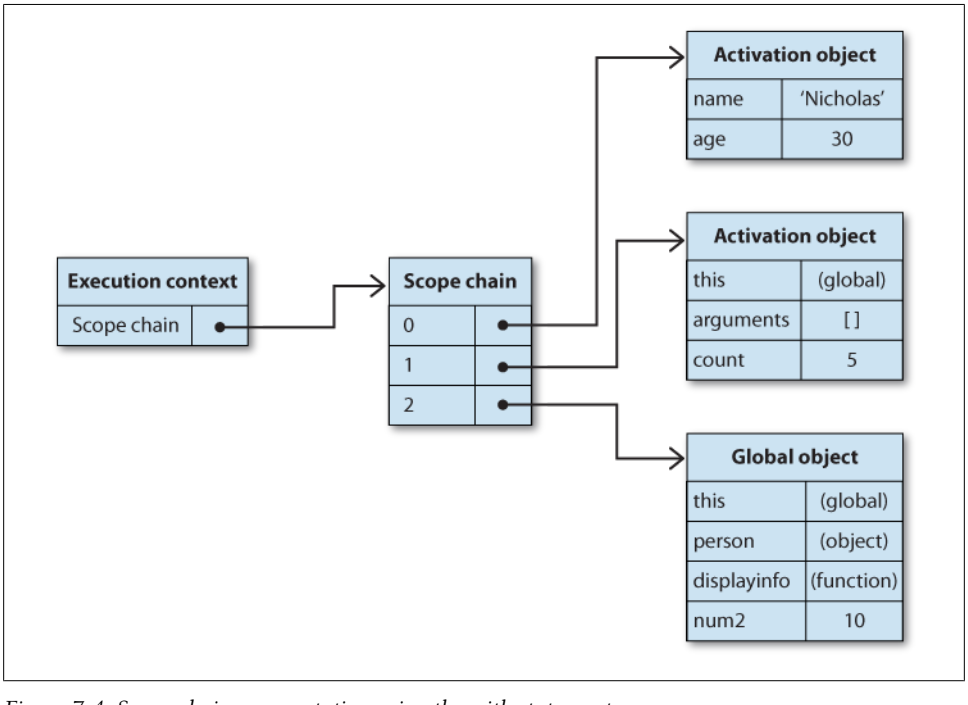


Figure 7-4. Scope chain augmentation using the with statement

Though it seems very convenient when an object's properties are being used repeatedly, this extra object in the scope chain hurts local identifier resolution. While code within a `with` statement is being executed, the local function variables now exist in the *second* object in the scope chain instead of the first, automatically slowing down identifier access. In the previous example, the `count` variable now takes longer to access because it's not in the first object of the scope chain. Once the `with` statement finishes executing, the scope chain is restored to its previous state. Due to this major downside, it's recommended to *avoid* using the `with` statement.

The second statement that augments the scope chain is the `catch` clause of a `try-catch` block. The `catch` clause behaves in a manner similar to the `with` statement where it adds a variable object to the front of the scope chain while it executes the code in the block. That variable object contains an entry for the named exception object specified by `catch`. However, the `catch` clause is executed only when an error occurs during execution of the `try` clause, making it somewhat less problematic than the `with` statement, though you should take care not to execute too much code within the `catch` clause to minimize the performance impact.

Minding scope chain depth is an easy way to get performance improvements with a small amount of work. Avoid unnecessarily augmenting the scope chain and inadvertently slowing down execution.

Efficient Data Access

Where data is stored in a script contributes directly to the amount of time it takes to execute. In general, there are four places from which data can be accessed in a script:

- Literal value
- Variable
- Array item
- Object property

Reading data always incurs a performance cost, and that cost depends on which of these four locations the data is stored in.

In most browsers, the cost of reading a value from a literal versus a local variable is so small as to be negligible; you should feel free to mix and match literals and local variables without worrying about a performance penalty. The real difference comes when you move to reading data from an array or object. Accessing values from one of these data structures requires a lookup of the location in which the data is stored, either by index (for array) or by property name (for objects).

To test the data access times based on data location, I created an experiment that reads values from each of these locations 200,000 times. You can find the experiment online at *http://www.nczonline.net/experiments/javascript/performance/data-access/*. The result of running this experiment on multiple browsers is that there is almost an even split across browsers as to which is faster: Internet Explorer, Opera, and Firefox 3 all access array items faster than object properties; Chrome, Safari, Firefox 2, and Firefox 3.1+ access object properties faster than array items (see Figure 7-5).

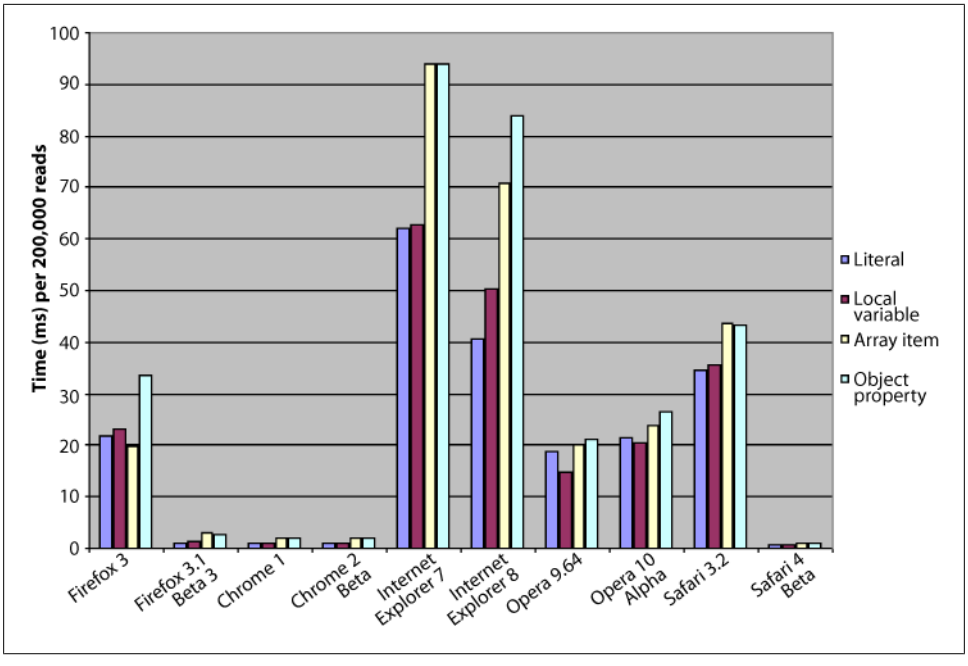


Figure 7-5. Data access time across browsers

The important lesson to take from this information is to always store frequently accessed values in a local variable. Consider the following code:

```
function process(data){
    if (data.count > 0){
        for (var i=0; i < data.count; i++){
            processData(data.item[i]);
        }
    }
}
```

This snippet accesses the value of `data.count` multiple times. At first glance, it looks like this value is used twice: once in the `if` statement and once in the `for` loop. In reality, though, `data.count` is accessed `data.count` plus 1 times in this function, since the control statement (`i < data.count`) is executed each time through the loop. The function will run faster if this value is stored in a local variable and then accessed from there:

```
function process(data){
    var count = data.count;
    if (count > 0){
        for (var i=0; i < count; i++){
            processData(data.item[i]);
        }
    }
}
```

The rewritten version of this function accesses `data.count` only once, at the beginning in order to store it in a local variable. The local variable `count` is used in its place elsewhere in the function, limiting the number of times an object property must be accessed to retrieve this value. This function will run faster than the previous function because the number of object property lookups has been reduced.

The effect of data access is exaggerated as the value's data structure depth increases. For example, `data.count` is faster to access than `data.item.count`, which is faster to access than `data.item.subitem.count`. When dealing with properties, the number of times a dot is used (for property lookup) directly relates to the amount of time it takes to access that value. Figure 7-6 shows the relative data access times by property depth across browsers. The tests for this research are part of the data access experiment located at *http://www.nczonline.net/experiments/javascript/performance/data-access/*.

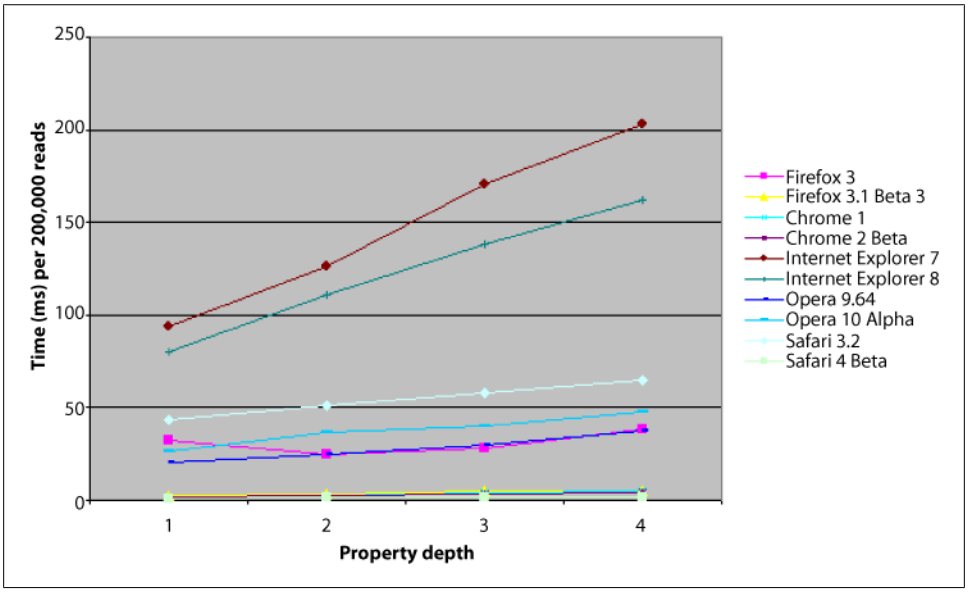


Figure 7-6. Access times for object properties by depth

A good approach to take when dealing with data access is to store in a local variable any object property or array item that is used more than once in a function.

For most browsers, there is virtually no difference between using dot notation for object property access (`data.count`) and bracket notation (`data["count"]`). The one exception is Safari, where bracket notation is significantly slower than dot notation. This holds true even for Safari 4 and later using the Nitro JavaScript engine.

Using local variables is especially important when dealing with `HTMLCollection` objects (those returned from DOM methods such as `getElementsByTagName` and properties such as `element.childNodes`). Each `HTMLCollection` object is actually a live query being run against the DOM document every time a property is accessed. For example:

```
var divs = document.getElementsByTagName("div");
for (var i=0; i < divs.length; i++){  //Avoid!
    var div = divs[i];
    process(div);
}
```

The first line of this code creates a query that returns every `<div>` element on the page and stores that query in `divs`. Each time `divs` has a property accessed either by name or by index, the DOM actually reexecutes that query against the entire page; in this code, it occurs each time `divs.length` or `divs[i]` is accessed. These property lookups take longer than the average non-DOM object property or array item lookup. It's therefore important to store such values in local variables whenever possible to avoid the requerying penalty associated with `HTMLCollection` objects. For example:

```
var divs = document.getElementsByTagName("div");
for (var i=0, len=divs.length; i < len; i++){  //Better
    var div = divs[i];
    process(div);
}
```

This example stores the length of the `divs HTMLCollection` in a local variable, limiting the number of times the object is accessed directly. In the previous version of this code, `divs` was accessed twice per iteration: once to retrieve the object in the given position, and once to check the length. This new version eliminates direct length-checking with each iteration.

Generally speaking, interacting with DOM objects is always more expensive than interacting with non-DOM objects. Due to DOM behavior, property lookups typically take longer than non-DOM property lookups. The `HTMLCollection` object is the worst-performing object in the DOM. If you need to repeatedly access members of an `HTMLCollection`, it is more efficient to copy them into an array first.

Flow Control

Next to data access, flow control is perhaps the most important aspect of JavaScript relating to performance. JavaScript, as with most programming languages, has a number of flow control statements that determine which part of the code should be executed next. There's a series of conditional and loop statements that enable developers to precisely control how execution flows from one part of the code to another. Choosing the right option at each point can dramatically affect how fast your script runs.

Fast Conditionals

The classic question of whether to use a `switch` statement or a series of `if` and `else` statements is not unique to JavaScript and has spurred discussions in nearly every programming language that has these constructs. The real issue is not between individual statements, of course, but rather relates to the speed with which each is able to handle a range of conditional statements. The details of this section are based on tests that you can run at *http://www.nczonline.net/experiments/javascript/performance/conditional-branching/*.

The if statement

Discussions usually begin surrounding complex `if` statements such as this:

```
if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 5){
    return result5;
} else if (value == 6){
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else if (value == 9){
    return result9;
} else {
    return result10;
}
```

Typically, this type of construct is frowned upon. The major problem is that the deeper into the statement the execution flows, the more conditions have to be evaluated. It will take longer to complete the execution when `value` is 9 than if `value` is 0 because every other condition must be evaluated beforehand. As the overall number of conditions increases, so does the performance hit for going deep into the conditions. While having a large number of `if` conditions isn't advisable, there are steps you can take to increase the overall performance.

The first step is to arrange the conditions in decreasing order of frequency. Since exiting after the first condition is the fastest operation, you want to make sure that happens as often as possible. Suppose the most common case in the previous example is that `value` will equal 5 and the second most common is that `value` will equal 9. In that case, you know five conditions will be evaluated before getting to the most common case

and nine before getting to the second most common case; this is incredibly inefficient. Even though the increasing numeric order of the conditions makes it easier to read, it should actually be rewritten as follows:

```
if (value == 5){
    return result5;
} else if (value == 9){
    return result9;
} else if (value == 0){
    return result0;
} else if (value == 1){
    return result1;
} else if (value == 2){
    return result2;
} else if (value == 3){
    return result3;
} else if (value == 4){
    return result4;
} else if (value == 6){
    return result6;
} else if (value == 7){
    return result7;
} else if (value == 8){
    return result8;
} else {
    return result10;
}
```

Now the two most common conditions appear at the top of the `if` statement, ensuring optimal performance for these cases.

Another way to optimize `if` statements is to organize the conditions into a series of branches, following a binary search algorithm to find the valid condition. This is advisable in the case where a large number of conditions are possible and no one or two will occur with a high enough rate to simply order according to frequency. The goal is to minimize the number of conditions to be evaluated for as many of the conditions as possible. If all of the conditions for `value` in the example will occur with the same relative frequency, the `if` statements can be rewritten as follows:

```
if (value < 6){

    if (value < 3){
        if (value == 0){
            return result0;
        } else if (value == 1){
            return result1;
        } else {
            return result2;
        }
    } else {
        if (value == 3){
            return result3;
        } else if (value == 4){
            return result4;
```

```
        } else {
            return result5;
        }
    }

} else {

    if (value < 8){
        if (value == 6){
            return result6;
        } else {
            return result7;
        }
    } else {
        if (value == 8){
            return result8;
        } else if (value == 9){
            return result9;
        } else {
            return result10;
        }

    }
}
```

This code ensures that there will never be any more than four conditions evaluated. Instead of evaluating each condition to find the right value, the conditions are separated first into a series of ranges before identifying the actual value. The overall performance of this example is improved because the cases where eight and nine conditions need to be evaluated have been removed. The maximum number of condition evaluations is now four, creating an average savings of about 30% off the execution time of the previous version. Keep in mind, also, that an `else` statement has no condition to evaluate. However, the problem remains that each additional condition ends up taking more time to execute, affecting not only the performance but also the maintainability of this code. This is where the `switch` statement comes in.

The switch statement

The `switch` statement simplifies both the appearance and the performance of multiple conditions. You can rewrite the previous example using a `switch` statement as follows:

```
switch(value){
    case 0:
        return result0;
    case 1:
        return result1;
    case 2:
        return result2;
    case 3:
        return result3;
    case 4:
        return result4;
    case 5:
```

```
            return result5;
        case 6:
            return result6;
        case 7:
            return result7;
        case 8:
            return result8;
        case 9:
            return result9;
        default:
            return result10;
    }
```

This code clearly indicates the conditions as well as the return values in an arguably more readable form. The `switch` statement has the added benefit of allowing fall-through conditions, which allow you to specify the same result for a number of different values without creating complex nested conditions. The `switch` statement is often cited in other programming languages as the hands-down better option for evaluating multiple conditions. This isn't because of the nature of the `switch` statement, but rather because of how compilers are able to optimize `switch` statements for faster evaluation. Since most JavaScript engines don't have such optimizations, performance of the `switch` statement is mixed.

Firefox handles `switch` statements very well, with each condition's evaluation executing in roughly the same amount of time regardless of the order in which they are defined. That means the case of `value` equal to 0 will take roughly the same amount of time to execute as when `value` is 9. Other browsers, however, aren't nearly as good. Internet Explorer, Opera, Safari, and Chrome all show noticeable increases in the execution time as you get deeper into the `switch` statement. Those increases, however, are smaller than the increases experienced with each additional condition of an `if` statement. You can therefore improve the performance of `switch` statements by ordering the conditions in decreasing rate of frequency (the same as `if` statement optimization).

In JavaScript, `if` statements are generally faster than `switch` statements when there are just *one or two conditions* to be evaluated. When there are more than two conditions, and the conditions are simple (not ranges), the `switch` statement tends to be faster. This is because the amount of time it takes to execute a single condition in a `switch` statement is often less than it takes to execute a single condition in an `if` statement, making the `switch` statement optimal only when there are a larger number of conditions.

Another option: Array lookup

There are more than two solutions for dealing with conditionals in JavaScript. Alongside the `if` statement and the `switch` statement is a third approach: looking up values in arrays. The example for this section maps a given number to a specific result, which is exactly what arrays are for. Instead of writing a large `if` statement or `switch` statement, you can use the following code:

```
//define the array of results
var results = [result0, result1, result2, result3, result4, result5, result6,
result7,
               result8, result9, result10]

//return the correct result
return results[value];
```

Instead of using conditional statements, all of the results are stored in an array whose index maps to the `value` variable. Retrieving the appropriate result is simply a matter of array value lookup. Although array lookup times also increase the deeper into the array you go, the incremental increase is so small that it is irrelevant relative to the increases in each condition evaluation for `if` and `switch` statements. This makes array lookup ideal whenever there are a large number of conditions to be met, and the conditions can be represented by discrete values such as numbers or strings (for strings, you can use an `Object` to store the results rather than an `Array`).

It's not practical to use array lookup for small numbers of results because array lookup is often slower than evaluating a small number of conditions. Array lookups can be very helpful when there are a large number of ranges because they eliminate the need to test both upper and lower bounds; you can simply fill in that range of indexes in the array with the appropriate value and do a straight array lookup.

The fastest conditionals

The three techniques presented here—the `if` statement, the `switch` statement, and array lookup—each have their uses in optimizing code execution:

- Use the `if` statement when:
  - There are no more than two discrete values for which to test.
  - There are a large number of values that can be easily separated into ranges.
- Use the `switch` statement when:
  - There are more than two but fewer than 10 discrete values for which to test.
  - There are no ranges for conditions because the values are nonlinear.
- Use array lookup when:
  - There are more than 10 values for which to test.
  - The results of the conditions are single values rather than a number of actions to be taken.

Fast Loops

As mentioned in Chapter 1, loops are a frequent source of performance issues in JavaScript, and the way you write loops drastically changes its execution time. Once again, JavaScript developers don't get to rely on compiler optimizations that make loops faster

regardless of the initial code, so it's important to understand the various ways to write loops and how they affect performance.

Simple loop performance boosts

There are four different types of loops in JavaScript. In this section, we will discuss three of them: the `for` loop, the `do-while` loop, and the `while` loop. (The fourth type is a `for-in` loop that is used to iterate over object properties, but I won't cover it here because its purpose is very unique.) The various loop types are coded as follows:

```
//unoptimized code
var values = [1,2,3,4,5];

//for loop
for (var i=0; i < values.length; i++){
    process(values[i]);
}

//do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < values.length);

//while loop
var k=0;
while (k < values.length){
    process(values[k++]);
}
```

Each of the loops in this example achieves the same result: all items in the `values` array are passed into the `process` function. These are the most common constructs used for iterating over a number of values in an array. Each of these loops runs in about the same amount of time because they're doing roughly the same amount of work. There are, however, ways to improve the performance.

Perhaps the most glaring issue in each loop is the constant comparison of the iterator variable against the array length. As mentioned earlier in this chapter, property lookup is a much more expensive operation than local variable access. This code is retrieving the value of `values.length` every time the loop executes to see whether the terminal condition has been reached. This is incredibly inefficient given that the length of the array won't change while the loop is being executed. Using a local variable instead of a property lookup can speed up the loops:

```
var values = [1,2,3,4,5];
var length = values.length;

//for loop
for (var i=0; i < length; i++){
    process(values[i]);
}
```

```
//do-while loop
var j=0;
do {
    process(values[j++]);
} while (j < length);

//while loop
var k=0;
while (k < length){
    process(values[k++]);
}
```

Each loop now uses the local variable `length` as its comparison point instead of `values.length`, eliminating a property lookup each time through the loop. This technique is especially important when dealing with `HTMLCollection` objects because, as mentioned previously, every property access on such an object is actually a query against the DOM for all nodes matching some criteria. That makes a property lookup on an `HTMLCollection` very expensive and, when included in the terminal condition of a loop, adds significant execution time to the overall loop.

Another simple way to improve the performance of a loop is to decrement the iterator toward 0 rather than incrementing toward the total length. Making this simple change can result in savings of up to 50% off the original execution time, depending on the complexity of each iteration. For example:

```
var values = [1,2,3,4,5];
var length = values.length;

//for loop
for (var i=length; i--;){
    process(values[i]);
}

//do-while loop
var j=length;
do {
    process(values[--j]);
} while (j);

//while loop
var k=length;
while (k--){
    process(values[k]);
}
```

Each of these loops is now even faster by virtue of changing the terminal condition to a comparison against 0 (note that the terminal condition evaluates to `true` once the iterator variable equals 0). The performance of each type of loop is comparable, so you needn't worry about choosing among the three variations for speed purposes.

Be careful when using the native `indexOf` method for arrays. This method can take significantly longer to iterate over each array item than using a regular loop. If speed is your primary concern, use one of the three loop types mentioned in this section.

Avoid the for-in loop

Another variation of the `for` loop is the `for-in` loop, whose purpose is to iterate over the enumerable properties of a JavaScript object. Typical usage is as follows:

```
for (var prop in object){
    if (object.hasOwnProperty(prop)){  //to filter out prototype properties
        process(object[prop]);
    }
}
```

This code iterates over the properties in a given object, using the `hasOwnProperty` method to ensure that only instance properties are processed.

Because the `for-in` loop has a specific purpose, there is little you can do to change its performance. The terminal condition cannot be altered, and the order of the properties to iterate over cannot be changed. Further, a `for-in` loop is typically much slower than any of the other loops because it requires resolving every enumerable property on a particular object. That, in turn, means the object's prototype and entire prototype chain must be examined to extract these properties. Traversing the prototype chain, just like traversing the scope chain, takes time and slows down the performance of the entire loop.

If you know the specific properties you're interested in, it's much faster to create a standard loop (`for`, `do-while`, or `while`) and iterate over an array of names, such as:

```
//known properties to iterate over
var props = ["name", "age", "title"];

//while loop
var i=props.length;
while (i--){
    process(object[props[i]]);
}
```

This loop runs much faster than the `for-in` loop, and not simply because of the small number of properties in the `props` array. Even increasing the number of properties over which to iterate would yield significantly better performance than the `for-in` loop. The loop in this example takes advantage of all the normal loop performance enhancements and still allows iteration over a known set of object properties.

Naturally, this approach works only when you know the object properties to iterate over; when dealing with unknown properties, as with JSON objects, a `for-in` loop may still be necessary.

Unrolling loops

It is a common practice in several programming languages to unroll small loops to improve performance. The basis of this practice is that limiting the number of iterations can mitigate the performance overhead of a loop. The implementation of such a solution is typically called *unrolling the loop*, which means making each iteration do the work of multiple iterations. Consider the following loop:

```
var i=values.length;
while (i--){
    process(values[i]);
}
```

If there are only five items in the `values` array, it is actually faster to remove the loop and do the work on each value individually:

```
//unrolled loop
process(values[0]);
process(values[1]);
process(values[2]);
process(values[3]);
process(values[4]);
```

Of course, this approach is arguably less maintainable, as it takes more code to write and any change to the number of items in the `values` array requires changes to the code. Further, the performance gains for such a small number of statements aren't worth the maintenance overhead. This technique can be quite useful, however, when you're dealing with a large number of values and a potentially large number of iterations.

Tom Duff, a computer programmer working for Lucasfilm at the time, first proposed a construct for unrolling loops in the C programming language. This pattern became known as Duff's Device and was later converted to JavaScript by Jeff Greenberg, who also published one of the first comprehensive studies on JavaScript performance optimization (which is still available at *http://home.earthlink.net/~kendrasg/info/js_opt/*). Greenberg's Duff's Device implementation is as follows:

```
var iterations = Math.ceil(values.length / 8);
var startAt = values.length % 8;
var i = 0;

do {
    switch(startAt){
        case 0: process(values[i++]);
        case 7: process(values[i++]);
        case 6: process(values[i++]);
        case 5: process(values[i++]);
        case 4: process(values[i++]);
        case 3: process(values[i++]);
        case 2: process(values[i++]);
        case 1: process(values[i++]);
    }
    startAt = 0;
} while (--iterations > 0);
```

The idea behind Duff's Device is that each trip through the loop does the work of between one and eight iterations of a normal loop. This is done by first determining the number of iterations by dividing the total number of array values by eight. Duff found that eight was an optimal number to use for this processing (it's not arbitrary). Since not all array lengths will be equally divisible by eight, you must also calculate how many items won't be processed by using the modulus operator. The `startAt` variable, therefore, contains the number of additional items to be processed. This variable is used only the first time through the loop, to do the extra work, and then is set back to zero so that each subsequent trip through the loop results in a full eight items being processed. Duff's Device runs faster than a normal loop over a large number of iterations, but it can be made even faster.

The book *Speed Up Your Site* (New Riders) introduced a version of Duff's Device in JavaScript that moves the processing of the extra array items outside the main loop, allowing the `switch` statement to be removed and resulting in an even faster way of processing a large number of items:

```
var iterations = Math.floor(values.length / 8);
var leftover = values.length % 8;
var i = 0;

if (leftover > 0){
    do {
        process(values[i++]);
    } while (--leftover > 0);
}

do {
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
    process(values[i++]);
} while (--iterations > 0);
```

This code executes faster over a large number of array items primarily due to the removal of the `switch` statement from the main loop. As discussed earlier in this chapter, conditionals do have performance overhead; removing that overhead from the algorithm speeds up the processing. The separation of processing into two discrete loops allows this augmentation.

Duff's Device, and the modified version presented here, is useful primarily with large arrays. For small arrays, the performance gain is minimal compared to standard loops. Therefore, you should attempt to use Duff's Device only if you notice a performance bottleneck relating to a loop that must process a large number of items.

String Optimization

String manipulation is a very common occurrence in JavaScript. There are multiple ways to deal with string processing, depending on the particular task, and each task brings with it specific performance considerations. There are a number of different ways to manipulate strings, whether it be using built-in string methods and operators or intermixing the use of regular expressions and arrays. The exact technique to use for optimal performance is tied directly to the type of manipulation being performed.

String Concatenation

Traditionally, string concatenation has been one of the poorest-performing aspects of JavaScript. Typically, string concatenation is done using the plus operator (`+`), such as in the following:

```
var text = "Hello";
text += " "
text += "World!";
```

Early browsers had no optimization for such operations. Since strings are immutable, that meant creating intermediate strings to contain the concatenation result. This constant creation and destruction of strings behind the scenes led to very poor string concatenation performance.

Having discovered this, developers turned to the JavaScript `Array` object for help. One of the `Array` object's methods is `join`, which concatenates all items in the array and inserts a given string between the items. Instead of using the plus operator, each string is added to an array and the `join` method is called when all items have been added. For example:

```
var buffer = [],
    i = 0;
buffer[i++] = "Hello";
buffer[i++] = " ";
buffer[i++] = "World!";

var text = buffer.join("");
```

In this code, each string is added into the `buffer` array. The `join` method is called after all strings are in the array, returning the concatenated string and storing it in the variable `text`. Adding the items directly into the appropriate index is slightly faster than calling `push` for each value. This technique proved to be much faster in early browsers than using the plus operator because no intermediate strings are being created and destroyed. However, browser string optimizations have changed the string concatenation picture.

Firefox was the first browser to optimize string concatenation. Beginning with version 1.0, the array technique is actually slower than using the plus operator in all cases. Other browsers have also optimized string concatenation, so Safari, Opera, Chrome,

and Internet Explorer 8 also show better performance using the plus operator. Internet Explorer prior to version 8 didn't have such an optimization, and so the array technique is always faster than the plus operator.

This doesn't necessarily mean browser detection should be used whenever string concatenation is necessary. There are two factors to consider when determining the most appropriate way to concatenate strings: the size of the strings being concatenated and the number of concatenations.

All browsers can comfortably complete the task in less than one millisecond using the plus operator when the size of the strings is relatively small (20 characters or less) and the number of concatenations is also relatively small (1,000 concatenations or less). There is no reason to consider anything other than using the plus operator if this is your situation.

As you increase the number of concatenations for small strings, or the size of the strings with a small number of concatenations, the performance gets significantly worse in Internet Explorer through version 7. Also, as the size of the strings increases, the performance difference between using the plus operator and the array technique decreases in Firefox. As the number of concatenations increases, the difference between the two techniques decreases in Safari as well. The only browsers where the plus operator remains consistently and significantly faster with varying string size and concatenation numbers are Chrome and Opera.

With all of the performance variance across browsers, the technique to use is heavily dependent on the use case as well as on the browsers you're targeting. If your users largely use Internet Explorer 6 or 7, it may be worth using the array technique all the time because that will affect the largest number of people. The performance decrease of the array technique in other browsers is typically much less than the performance increase gained in Internet Explorer, so try to balance your users' experience based on their browsers rather than trying to target specific situations and browser versions. In most common cases, however, using the plus operator is preferred.

Trimming Strings

One of the most glaring omissions of JavaScript strings is the lack of a native trim method used to remove leading and trailing whitespace. The most common implementation of a `trim` function is as follows:

```
function trim(text){
    return text.replace(/^\s+|\s+$/g, "");
}
```

This implementation uses a regular expression that matches one or more whitespace characters at the beginning or end of the string. The string's `replace` method is used to replace any matches with an empty string. This implementation, however, has a performance issue based in the regular expression.

The performance impact comes from two aspects of the regular expression: the pipe operator, indicating that there are two patterns to match, and the `g` flag, indicating that the pattern should be applied globally. Taking this into mind, you can rewrite the function to be a bit faster by breaking up the regular expression into two and getting rid of the `g` flag:

```
function trim(text){
    return text.replace(/^\s+/, "").replace(/\s+$/, "");
}
```

Breaking the single `replace` method into two calls allows each regular expression to become much simpler and, therefore, faster. This method is faster than the original, but you can optimize it even further.

Steven Levithan, after performing research on the fastest way to execute string trimming in JavaScript, arrived at the following function:

```
function trim(text){
    text = text.replace(/^\s+/, "");
    for (var i = text.length - 1; i >= 0; i--) {
        if (/\S/.test(text.charAt(i))) {
            text = text.substring(0, i + 1);
            break;
        }
    }
    return text;
}
```

This `trim` function consistently performs better than other variations. The source of the speed is keeping the regular expressions as simple as possible. The first line removes leading whitespace and then the `for` loop is used to strip trailing whitespace. The loop uses a very simple, single-character regular expression that matches nonwhitespace characters. This information is used to either remove a character from the string or break the loop. The resulting function performs faster than the previous versions across all browsers. For Levithan's complete analysis, see his post at *http://blog.stevenlevithan.com/archives/faster-trim-javascript*.

As with string concatenation, the speed of string trimming matters only if it is performed with enough frequency during execution. The second `trim` function in this section performs fine for smaller strings over the course of a few calls; the third `trim` function is significantly faster when used on longer strings.

The next version of the ECMAScript specification, code-named ECMAScript 3.1, defines a native `trim` method for strings; it is likely that this native version will be faster than any of the functions in this section. When available, the native function should be used.

Avoid Long-Running Scripts

One of the critical performance issues with JavaScript is that code execution freezes a web page. Because JavaScript is a single-threaded language, only one script can be run at a time per window or tab. This means that all user interaction is necessarily halted while JavaScript code is being executed. This is an important feature of browsers since JavaScript may change the underlying page structure during its execution, with the possibility of nullifying or altering the response to user interaction.

If JavaScript code isn't carefully crafted, it's possible to freeze the web page for an extended period of time and ultimately cause the browser to stop responding. Most browsers will detect long-running scripts and notify the user of a problem with a dialog box asking whether the script should be allowed to proceed.

Exactly what causes the browser to display the long-running script dialog varies depending on the vendor:

- Internet Explorer monitors the number of statements that have been executed by a script. When a maximum number of statements have been executed, 5 million by default, the long-running script dialog is displayed (as shown in Figure 7-7).
- Firefox monitors the amount of time a script is taking to execute. When a script takes longer than a preset amount of time, 10 seconds by default, the long-running script dialog is displayed.
- Safari also uses the execution time to determine whether a script is long-running. The default timeout is set to five seconds, after which the long-running script dialog is displayed.
- Chrome as of version 1.0 has no set limit on how long JavaScript is allowed to run. The process will crash when it has run out of memory.
- Opera is the only browser that doesn't protect against long-running scripts. Scripts are allowed to continue until execution is complete.

Figure 7-7. Internet Explorer 7 long-running script dialog

If you ever see the long-running script dialog, it's an indication that the JavaScript code needs to be refactored. Generally speaking, no single continuous script execution should take longer than 100 milliseconds; anything longer than that and the web page will almost certainly appear to be running slowly to the user. Brendan Eich, the creator of JavaScript, is also quoted as saying, "[JavaScript] that executes in whole seconds is probably doing something wrong...."

The most common reasons why a script takes too long to execute include:

Too much DOM interaction
: DOM manipulation is more expensive than any other JavaScript process. Minimizing DOM interactions significantly cuts the JavaScript runtime. Most browsers update the DOM only after the entire script has finished executing, which slows the perceived responsiveness of the web page to the user.

Loops that do too much
: Any loop that either runs too many times or performs too many operations with each iteration can cause long-running script issues. It helps separate out functionality whenever possible. The effect is worsened when loops are used to perform DOM manipulations, sometimes causing the browser to completely freeze without ever showing the long-running script dialog.

Too much recursion
: JavaScript engines put a limit on the amount of recursion that scripts can use. Rewriting the code to avoid recursion helps ameliorate the issue.

Sometimes simple code refactoring, keeping these issues in mind, can prevent runaway scripts. There may, however, be times when complex processes must necessarily be executed for the web application to function correctly. In that case, the code must be restructured to yield periodically, as explained in the next section.

Yielding Using Timers

The single-threaded nature of JavaScript means that only one script can be executed in a window or tab at any given point in time. No user interactions can be processed during this time and so it's necessary to introduce breaks in long-executing JavaScript code. On simple web pages, the breaks occur naturally as the user interacts with the page. In complex web applications, it's often necessary to insert the breaks yourself. The easiest way to do this is to use a timer.

Timers are created using the `setTimeout` function, passing in the function to execute as well as a delay (in milliseconds) before the function should be executed. When the delay has passed, the code to execute is placed into a queue. The JavaScript engine uses this queue to determine what to do next. When a script finishes executing, the JavaScript engine yields to allow other browser tasks to catch up. The web page display is typically updated during this time in relation to changes made via the script. Once the display has been updated, the JavaScript engine checks for more scripts to run on the

queue. If another script is waiting, it is executed and the process repeats; if there are no more scripts to execute, the JavaScript engine remains idle until another script appears in the queue.

When you create a timer, you're actually scheduling some code to be inserted into the JavaScript engine's queue to be executed later. That insertion happens after the amount of time specified when calling `setTimeout`. In essence, timers push code execution off into the future, where all long-running script limits are reset. Consider the following code:

```
window.onload = function(){

    //Page Load

    //create first timer
    setTimeout(function(){

        //Delayed Script 1

        setTimeout(function(){

            //Delayed Script 2

        }, 100);

        //Delayed Script 1, continued

    }, 100);

};
```

In this example, a script is run when the page loads. That script calls `setTimeout` to create the first timer. When that timer executes, it calls `setTimeout` again to create a second timer. The second delayed script cannot start running, however, until the first has finished executing and the browser has updated the display. Figure 7-8 shows the timeline for this code execution, indicating that no two scripts are run at the same time.

Timers are the de facto standard for splitting up JavaScript code execution in browsers. Whenever a script is taking too long to complete, look to delay parts of the execution until later.

Note that very small timer delays can also cause the browser to become unresponsive. It's recommended to never use a delay of zero milliseconds, as this isn't enough time for all browsers to properly update their display. In general, delays between 50 and 100 milliseconds are appropriate and allow browsers enough idle time to perform necessary display updates.

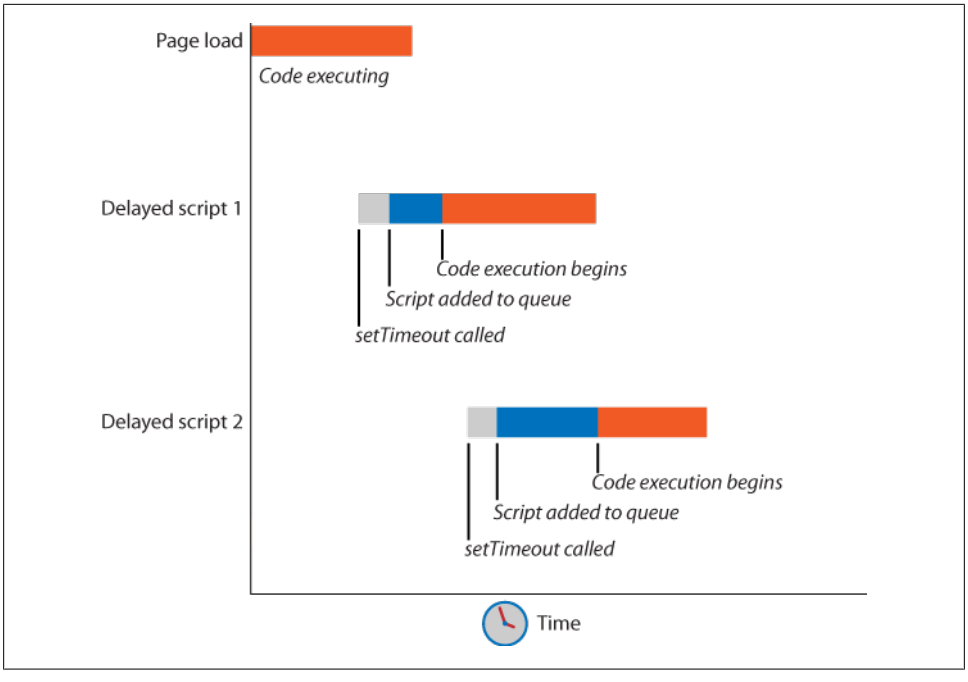


Figure 7-8. JavaScript code execution with timers

Timer Patterns for Yielding

Array processing is one of the most frequent causes of long-running scripts. Typically, this is because processing must be done on each member of the array, and so the execution time increases directly in proportion to the number of items in the array. If the array processing doesn't have to be executed synchronously, it is a good candidate for splitting up using timers.

In my book, *Professional JavaScript for Web Developers*, Second Edition (Wrox), I describe a simple function that can be used to split up the processing of arrays using timers:

```
function chunk(array, process, context){
    setTimeout(function(){
        var item = array.shift();
        process.call(context, item);

        if (array.length > 0){
            setTimeout(arguments.callee, 100);
        }
    }, 100);
}
```

The chunk function accepts three arguments: an array of data to process, a function with which to process each item, and an optional context argument in which the processing function should be executed (by default, all functions passed into setTimeout are run in the global context, so this is equal to window). Processing of the items is done using timers, and so the code execution yields after each item has been processed. The next item to process is always at the front of the array and is removed before being processed. Afterward, a check is made to determine whether there are any further values left to process. If so, a new timer is created and the function is called again via arguments.callee. Note that the chunk function uses the passed-in array as a "to do" list of items to process and so is altered once execution is complete. You can use the function as follows:

```
var names = ["Nicholas", "Steve", "Doug", "Bill", "Ben", "Dion"],
    todo = names.concat();  //clone the array

chunk(todo, function(item){
    console.log(item);
});
```

The code in this simple example outputs each name in the names array to the console (available in Firefox with Firebug installed, Internet Explorer 8+, Safari 2+, and all versions of Chrome). The processing function is very short but could easily be replaced with something more complex. The chunk function is best used with long arrays where each item requires significant processing.

Another popular pattern is to perform small, sequential parts of a larger operation using timers. Julien Lecomte presented this pattern in his blog post, "Running CPU Intensive JavaScript Computations in a Web Browser", in which he showed how sorting of a large data set could be achieved using an inefficient algorithm (bubble sort) without incurring a long-running script issue. The following is an adaptation of Lecomte's code:

```
function sort(array, onComplete){

    var pos = 0;

    (function(){

        var j, value;

        for (j=array.length; j > pos; j--){
            if (array[j] < array[j-1]){
                value = data[j];
                data[j] = data[j-1];
                data[j-1] = value;
            }
        }

        pos++;

        if (pos < array.length){
            setTimeout(arguments.callee,10);
```

```
            } else {
                onComplete();
            }

        })();

    }
```

The `sort` function splits up each traversal through the array for sorting, allowing the browser to continue functioning while this processing occurs. The inner anonymous function is called immediately to do the first traversal and then is called subsequently via a timer by passing `arguments.callee` into `setTimeout`. When the array is finally sorted, the `onComplete` function is called to notify the developer that the data is ready to be used. The function can be used as follows:

```
sort(values, function(){
    alert("Done!");
});
```

When sorting an array with a large number of items, the difference in browser responsiveness is immediately apparent.

Summary

The speed with which JavaScript executes is very dependent on how it is written. In this chapter, you learned several ways to speed up JavaScript code execution:

- Managing your scope is important, since out-of-scope variables take longer to access than local variables. Try to avoid constructs that artificially augment the scope chain, such as the `with` statement and the `catch` clause of a `try-catch` statement. If an out-of-scope value is being used more than once, store it in a local variable to minimize the performance penalty.
- The way you store and access data can greatly impact the performance of your script. Literal values and local variables are always the fastest; you incur a performance penalty for accessing array items and object properties. If an array item or object property is used more than once, store it in a local variable to speed up access to the value.
- Flow control is also an important determinant of script execution speed. There are three ways to handle conditionals: the `if` statement, the `switch` statement, and array lookup. The `if` statement is best used with a small number of discrete values or a range of values; the `switch` statement is best used when there are between 3 and 10 discrete values to test for; array lookup is most efficient for a larger number of discrete values.
- Loops are frequently found to be bottlenecks in JavaScript. To make a loop the most efficient, reverse the order in which you process the items so that the control condition compares the iterator to zero. This is far faster than comparing a value

to a nonzero number and significantly speeds up array processing. If there are a large number of required iterations, you may also want to consider using Duff's Device to speed up execution.

- Be careful when using `HTMLCollection` objects. Each time a property is accessed on one of these objects, it requires a query of the DOM for matching nodes. This is an expensive operation that can be avoided by accessing `HTMLCollection` properties only when necessary and storing frequently used values (such as the `length` property) in local variables.
- Common string operations may have unintended performance implications. String concatenation is much slower in Internet Explorer than in other browsers, but it's not worth worrying about unless you're dealing with more than 1,000 concatenations. You can optimize string concatenation in Internet Explorer by using an array to store the individual strings and then calling `join()` to merge them together. Trimming strings may also be expensive, depending on the size of the string. Make sure to use the most optimal algorithm if trimming is a large part of your script.
- Browsers have limits on how long JavaScript can run, capping either the number of statements or the amount of time the JavaScript engine is allowed to run. You can circumvent these limits, and prevent the browser from displaying a warning about the long-running script, by using timers to split up the amount of work.

CHAPTER 8
Scaling with Comet

Dylan Schiemann

Sometimes Ajax just isn't fast enough.

When data needs to be asynchronously sent from the server to the client, Ajax alone is often inadequate. *Comet* is a catchall term describing the collection of techniques, protocols, and implementations that address making low-latency data transit to the browser both viable and scalable. Comet is not an acronym, but a humorous play on the term *Ajax* coined by Alex Russell.*

Goals of Comet include delivering data from the server to the client at any time, improving speed and scalability over traditional Ajax, and developing event-driven web applications.

Ajax and the introduction of background HTTP requests are clearly the defining technology that enables the performance possible in today's web applications. However, browsers and the traditional request/response pattern used in HTTP are ill-equipped to scale to the needs of more demanding real-time applications such as chat, financial information, and document collaboration. All of these applications require low-latency data transit to deliver on user experience expectations.

In this chapter, I'll briefly cover how Comet works, and I'll discuss the techniques that are common today and the performance pros and cons of each. I'll conclude with solutions to cross-domain Comet and to other web application implementation issues when using Comet techniques.

How Comet Works

Comet works by taking advantage of less commonly used features of the HTTP specification. Through the more intelligent management of longer-lived connections, and by reducing the server-side resources per connection, Comet can easily provide more

* Both Ajax and Comet live under the kitchen sink.

simultaneous connections than a traditional web server, and faster data transit between the client and the server.

Large-scale applications must use asynchronous connection handling because traditional server architectures require the use of one thread per connection. For high-concurrency applications, Comet servers generally leverage event libraries such as libevent,[†] epoll,[‡] and kqueue,[§] depending on the operating system. Operating systems handle asynchronous I/O in various ways, the traditional method being `select` or `poll`. Your application can use these constructs to ask the operating system which sockets are ready to be written to or read from, to avoid ever incurring a blocking read or write.

What if the scale of your application is not large, but you want the benefits of Comet? Even a site of 50,000 visits per day with a typical connection time of three minutes averages only 92 open connections. Although you may need to raise the max thread count on your server, 92 threads is not a terrible approach for smaller but high-performance web sites.

The use of one thread per connection for high-performance Comet-based sites is problematic, so most Comet servers either significantly reduce the resource overhead per thread, or make use of microthreads or processes. For example, ErlyComet[‖] is written in Erlang, which is a virtual machine and microthreads-based functional language. Because a connection is represented by a process, and Erlang's event-driven approach makes it easy for processes to communicate with each other via message passing, Erlang makes it very easy to scale the number of connections, even on different servers.

By contrast, PHP makes for a very poor choice as a Comet server language because of its threading model, so most PHP web applications that wish to use Comet make use of an *off-board* approach.[#] To make this work, a Comet client is written in PHP that communicates with the Comet server written in another language. While programming languages for Comet servers in general do not matter (there is no shortage of attempts at PHP Comet servers), languages such as C, Erlang, and Python are better suited for creating a Comet server, and there are a number of great Comet servers written in Java as well. The term *on-board* is used when your web server is the same as your Comet server.

While on-board Comet provides the benefits of simplicity and often lives on the same domain, off-board Comet is much more common for larger-scale web sites, or for sites where the primary development language is not well suited to Comet performance. For

[†] *http://monkey.org/~provos/libevent/*

[‡] *http://linux.die.net/man/4/epoll*

[§] *http://people.freebsd.org/~jlemon/papers/kqueue.pdf*

[‖] *http://code.google.com/p/erlycomet/*

[#] *http://cometdaily.com/2008/05/22/on-board-vs-off-board-comet/*

example, a site such as Facebook would probably use an off-board solution for its chat application, whereas a site such as Meebo uses an on-board solution since virtually all of its site traffic uses Comet techniques.

On the client side, the common techniques include polling, long polling, forever frame (iframe), XHR streaming, and soon, WebSocket. In conjunction with these techniques for establishing a Comet connection, a number of protocols exist for sending messages between the client and the server. A toolkit such as the Dojo Toolkit, or a library such as js.io, can handle many of these complexities for you automatically, but understanding how these techniques work without a toolkit is essential to understanding how to evaluate and optimize Comet performance.

Transport Techniques

I'm now going to walk you through four different approaches to implementing the low-latency data communications that are the foundation of Comet: polling, long polling, forever frame, and XHR streaming.

Polling

Communication can easily become blocked or deadlocked in many browsers because of the limit on the maximum number of simultaneous connections allowed per server (see Chapter 11). The naïve approach that developers first take to solve the limit of connections is simple *polling*, where a web site or application makes a request every *x* milliseconds to check whether there are new updates to display in the user interface. A very simple polling example might look something like this:*

```
setTimeout(function(){xhrRequest({"foo":"bar"})}, 2000);

function xhrRequest(data){
    var xhr = new XMLHttpRequest();
    // handle the data to send it as parameters on the request
    xhr.open("get", "http://localhost/foo.php", true);
    xhr.onreadystatechange = function(){
        if(xhr.readyState == 4){
            // handle update from server
        }
    };
    xhr.send(null);
}
```

Simple polling is the least optimized but simplest Comet technique.

* For simplicity, we are ignoring extra handling for legacy browsers and error handling, but most JavaScript libraries provide an Ajax/XHR request function, so your working code may be different.

Long Polling

Polling is workable when messages are generated server side at known intervals. For instance, in a stock tracking application when new price updates are available on the server every five seconds, the polling interval on the browser can be matched to ensure that there is always one request per data element. Otherwise, HTTP requests are wasted and consume valuable CPU time and bandwidth. But polling can cause serious issues, even in cases when the data interval is known but the server is overloaded. Consider the case when the server hasn't yet responded to the previous request for data; now, a second or even third request is sent, bombarding the server with useless additional requests. Of course, you can change the polling interval to poll five seconds after each successful request, but there are much better Comet techniques than polling.

A far more adaptive method is *long polling*, where the browser makes a request to the server, and the server responds only when it has new data available. To support long polling, the server ends up holding on to a large collection of unanswered requests and their corresponding connections. The server "holds on" to the request's connection by returning a *Transfer-Encoding: chunked* or *Connection: close* response. When data is ready for a particular client or set of clients, those connections are identified and a response containing the payload is sent back to the browser. The browser immediately makes a request back to the server. If the connection drops, the client will attempt to reestablish a connection with the server. Although the request/response cycle is client-initiated, as with polling, all data flow occurs on the server's schedule rather than on the client's, allowing a more perfect approximation of the server→client data flow. Additionally, server oversaturation isn't as large a concern because the client won't make additional requests until after the server actually responds. Long polling became a mainstream technique with the introduction of the web-based chat client Meebo.

A typical implementation of the long-polling Comet technique involves the use of a Comet client, typically but not always written in JavaScript, and a Comet server, with versions available in almost every language. So, how do you create a Comet client? Let's examine a plain-vanilla long-polling example:

```
function longPoll(url, callback) {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            // send another request to reconnect to the server
            callback(xhr.responseText);
            xhr.open('GET', url, true);
            xhr.send(null);
        }
    }
    // connect to the server to open a request
    xhr.open('POST', url, true);
    xhr.send(null);
}
```

In the `longPoll` method, after creating an `XMLHttpRequest` with the given URL, we define what to do when the `readyState` of the XHR is 4 and data is returned. In this case, we open a new connection and send the response to a callback function listening for new data to be returned.

This approach solves the most common use case from the browser perspective: conserving available requests. It also gets rid of a large number of useless requests, and feels more instantaneous to the user.†

However, both polling and long polling introduce a new problem for traditional web servers that are not optimized to handle large numbers of long-held or long-lived connections, but rather are optimized to open and close connections as quickly as possible. Apache, for example, is designed to handle approximately 10,000 simultaneous connections per server, whereas a good Comet server should be able to handle more than 50,000 long-held connections to be cost-effective in delivering real-time applications. Fortunately, a number of viable commercial and open source servers exist today to address this problem. The Comet Maturity Guide compares the quality of a number of options on the market today.

There are other approaches to optimizing polling and long polling. For example, Meebo implements a hybrid of long polling and polling, with a contracted maximum duration of a connection that both the client and the server adhere to, making it easier to reestablish failed connections. Others have implemented a technique called *smart polling*, which is polling that has a decrease in the frequency of a request when data is not received. For example, you might poll every second when data is being returned, but for each request that receives an empty response, you may delay each subsequent request by a factor of 1.5 (e.g., 1s, 1.5s, 2.25s, etc.). Finally, if you have a long-polling connection open and you need to make an XHR, you can always abort the long-polling XHR to free a connection, and then restart the long-polling connection once the non-Comet XHR is complete.

It's important to choose the right alternative if you run into limitations in the number of connections available in the browser, or if you experience excessive load on your Comet server.

Forever Frame

While long polling is the most common technique in use today, Comet originally spawned from the *forever-frame* technique, where a hidden iframe is opened and the request is made for a document that relies on HTTP 1.1's *chunked encoding*. Chunked encoding was designed for the incremental loading of very large documents, so you can think of the forever-frame technique as a very large document that is incrementally written to. A very simple example of forever frame is as follows:

† *http://cometdaily.com/2007/11/06/comet-is-always-better-than-polling/*

```
function foreverFrame(url, callback) {
    var iframe = body.appendChild(document.createElement("iframe"));
    iframe.style.display = "none";
    iframe.src = url + "?callback=parent.foreverFrame.callback";
    this.callback = callback;
}
```

And a series of messages from the server might look like this:

```
<script>
parent.foreverFrame.callback("the first message");
</script>
<script>
parent.foreverFrame.callback("the second message");
</script>
```

Various browser hacks are necessary to invoke incremental rendering, such as adding a `<br />` element or a few kilobytes of whitespace after each script block is sent with data wrapped in a function call to the parent Comet client. (See Chapter 12 for more information about chunked encoding and browser exceptions.) To keep the iframe document from becoming very large in terms of file size, one optimization is to remove nodes from the iframe document after they are parsed.

The forever-frame technique was initially doomed within Internet Explorer because of a rather annoying user experience: the constant clicking sound of a page load completing. Internet Explorer treats each chunked encoding event as a page load. Gmail Talk popularized the essential workaround for this problem through the use of the `htmlfile` ActiveX object (*http://msdn2.microsoft.com/en-us/library/Aa752574.aspx*), making the forever-frame technique a viable solution. Here's a fragment on a solution for Internet Explorer:

```
function foreverFrame(url, callback){
    // http://cometdaily.com/2007/11/18/ie-activexhtmlfile-transport-part-ii/
    // note, do not use 'var tunnel...'
    htmlfile = new ActiveXObject("htmlfile");
    htmlfile.open();
    htmlfile.write(
        "<html><script>" +
        "document.domain='" + document.domain + "';" +
        "</script></html>");
    htmlfile.close();
    var ifrDiv = tunnel.createElement("div");
    htmlfile.body.appendChild(ifrDiv);
    ifrDiv.innerHTML = "<iframe src='" + url + "'></iframe>";
    foreverFrame.callback = callback;
}
```

`foreverFrame` creates, opens, and writes an HTML document into an `htmlfile` object, and sets the `document.domain` variable, which is essential for cross-subdomain Comet, or the more common case of the Comet server running on a different port than your normal web server. An iframe is then created inside the `htmlfile`'s body, and this iframe document is then used for your Comet connection. Using this technique, Internet

Explorer no longer plays a click event and its accompanying sound. Garbage collection may prevent the cleanup and removal of the connection, so an `onunload` function is necessary to remove the reference to `htmlfile` and explicitly call the garbage collector:

```
function foreverFrameClose() {
    htmlfile = null;
    CollectGarbage();
}
```

XHR Streaming

The cleanest API for communication with the server is through an `XMLHttpRequest`, since it provides direct access to the response text and headers, and this is normally the transport mechanism used for polling and long polling. Several browsers provide support for streaming through XHR, including Firefox, Safari, Chrome, and Internet Explorer 8. Like the forever-frame technique, XHR streaming allows successive messages to be sent from the server without requiring a new HTTP request after each response.

While the lack of support for streaming in Internet Explorer versions 7 and earlier precludes complete reliance on a streaming-based protocol, we can certainly leverage streaming when it is available to improve performance. When available, XHR streaming is currently the best-performing Comet transport in the browser since it does not require the overhead of an iframe or script tags (as the forever-frame technique does), and can continuously utilize a single HTTP response (which long polling doesn't do). While it is unfortunate that Internet Explorer does not support it, XHR streaming is still a valuable progressive enhancement. Users can upgrade browsers and instantly enjoy the benefit of improved performance.

XHR streaming is achieved with a standard `XMLHttpRequest`, but you can listen for `onreadystatechange` events with a `readyState` of 3 to access data that has been sent from the server (prior to the response being finished; that is, a `readyState` of 4), which allows you to handle data as it is received, without waiting for the connection to close:

```
function xhrStreaming(url, callback){
    xhr = new XMLHttpRequest();
    xhr.open('POST', url, true);
    var lastSize;
    xhr.onreadystatechange = function(){
        var newTextReceived;
        if(xhr.readyState > 2){
            // get the newest text
            newTextReceived =
             xhr.responseText.substring(lastSize);
            lastSize = xhr.responseText.length;
            callback(newTextReceived);
        }
        if(xhr.readyState == 4){
            // create a new request if the response is finished
            xhrStreaming(url, callback);
```

```
            }
        }
        xhr.send(null);
    }
```

While XHR streaming certainly opens the door to more efficient network utilization and reduced resource consumption for both client and server, you should be aware that in certain situations, streaming can actually negatively impact server efficiency. Some servers, when used in long-polling situations, defer the allocation of socket buffers until a response is ready and can almost immediately dispose of the buffer since the response will be finished as soon as it is sent. With streaming, these buffers are created and must be maintained for the life of the connection. Of course, this is a matter of how the server is optimized, and different servers perform differently.

On the client side, XHR streaming can potentially cause performance issues. If a streaming response is continued for too long, the browser suffers from excess memory usage. Several thousand successive messages on a single response can bring Firefox to its knees. You can easily correct this issue by simply finishing the response after each 100 messages (or after a byte limit, such as 50 KB), and creating a fresh new request (as long polling does for each message) for subsequent messages.

One of the added responsibilities when using XHR streaming is message partitioning. The browser receives a stream of text from the server, but must pull out the individual messages. Firefox supports a special content type, *multipart/x-mixed-replace*, which you can use to separate messages within the stream.‡ However, this is not widely supported, and as it turns out, you can write a JavaScript parser that pulls out individual messages and is actually faster than Firefox's multipart handler.

Future Transports

Work is currently being done in the HTML 5 working group on WebSocket,§ which would provide a web-safe *TCP socket* to greatly simplify the approach to tunneling from the client to the server. WebSocket would likely replace all forms of Comet connection techniques if it becomes widely adopted by browser vendors in a performant manner.

If WebSocket is adopted across the major browsers and has the expected performance metrics, it would quickly replace the other transport techniques described in this section.

Cross-Domain

It is worth noting that long polling does not support cross-domain requests if the browser does not support cross-domain XHR, but the forever-frame technique does at

‡ *http://cometdaily.com/2008/01/17/proposal-for-native-comet-support-for-browsers/*

§ *http://cometdaily.com/2008/07/04/html5-websocket/*

least support cross-subdomain. We can also get cross-subdomain XHR with various workarounds such as the one by Abe Fettig,‖ or in modern browsers that support cross-domain XHR, or with HTML 5's `postMessage`.#

XHR traditionally has a more restrictive security model than iframes or script tags that are included or inserted into a document. Thus, another option exists under the moniker *callback polling* or *JSONP polling*, which allows cross-domain polling through the insertion of script tags for each new request rather than relying on the XHR. This technique relies on the JSONP* technique for establishing implicit trust across domains. JSONP simply wraps the response from the server into a user-provided function, which then gets called with the return data. JSONP is not the be-all and end-all of security, but it establishes the same level of trust you would get from adding a script reference to a third-party domain.

JSONP works by returning data in a script that is evaluated, and the name of the function is specified in the request made to the server using `<script>` blocks instead of XHR. Support for cross-domain Comet is important for several reasons: requests to different domains don't count against the *two-connection* limit,† connections can be made to retrieve data from third-party services, and your Comet server can run on a separate server from your HTTP server, allowing for separate Comet-optimized servers and traditional HTTP-optimized servers (traditional servers are often suboptimal for Comet and vice versa).

The following example shows usage of this technique, which allows you to return data from another domain back to your current domain, by using the implicit trust of this technique:

```
function callbackPolling(url, callback){
    // create a script element that will load the response from the server
    var script = document.createElement("script");
    script.type = "text/javascript";
    script.src = url + "callback=callbackPolling.callback";
    callbackPolling.callback = function(data){
        // send a new request to wait for the next server-sent message
        callbackPolling(url, callback);
        // call the callback
        callback(data);
    };
    // add the element to initiate loading
    document.getElementsByTagName("head")[0].appendChild(script);
}
```

‖ *http://www.fettig.net/weblog/2005/11/30/xmlhttprequest-subdomain-update/*

http://www.whatwg.org/specs/web-apps/current-work/#crossDocumentMessages

* *http://ajaxian.com/archives/jsonp-json-with-padding*

† This is primarily an issue for Internet Explorer 6 and 7. Browser connection limits are explained in Chapter 11.

It is important to note that in Firefox, successive script additions are always evaluated in order for any given page. Consequently, if you are using this technique to wait for a response from the server and in the meantime you wish to make another JSONP request in the same frame/page, you won't receive a response until the first script is evaluated, which could take an indefinite amount of time since it is waiting for a message from the server. To overcome this issue, you can create separate iframes to encompass each JSONP request. With each request in a separate frame, the responses can be evaluated in parallel as soon as they are received.

Effects of Implementation on Applications

Our goals with client-side Comet performance are to reduce latency of data transit, conserve and manage HTTP connections, route messages, and handle cross-domain issues. On the server side, performance optimizations are made by conserving and sharing the number of HTTP connections, and by minimizing the memory, CPU, I/O, and bandwidth requirements for each connection.

Managing Connections

Servers will keep an HTTP connection open indefinitely for each user, resulting in many open connections even if data is minimal. There are two constraints with connections: memory and CPU. No matter what, each connection is going to incur some memory overhead from the OS and our language. If we use one thread (or process) per connection, we incur an entire execution stack memory overhead, typically 2 MB, though this can be lowered until it is almost reasonable. Additionally, as our thread count increases past our processor count, we will end up with *thrashing*, where our operating system spends more cycles switching threads on to processors than it spends executing our actual code. For this reason, we need to opt for an asynchronous network architecture.

The problem with `select` or `poll` is that these methods cause the operating system to examine each and every socket you have open to determine which ones are ready. This means that a call to `select`, even one that reveals that no sockets are ready to be read from, is cheap when there are few sockets, but takes an increasing amount of CPU time as the number of sockets increases. We can avoid this problem by using alternative techniques that avoid this $O(n)$ examination of sockets, such as kqueue on FreeBSD/OS X, epoll on Linux, and completion ports on Windows. There are network libraries in most major languages that wrap these details into coherent, cross-platform APIs, such as libevent in C, java.nio, and Twisted Python.

Performance optimization techniques vary widely based on usage scenarios. For example, consider chat, which typically has many users connected but only a few of them receiving data at any given time. In this case, being able to manage a large number of idle connections through server-side sharing of connections is useful. The web sites Orbited and Willow Chat are highly optimized for this usage scenario.

In other examples, such as a real-time stock quote monitoring application, many connections are updated constantly and few idle connections exist. Jetty, Lightstreamer, and Liberator are optimized for this case.

Measuring Performance

Measuring Comet performance has been discussed in numerous places and alluded to throughout this chapter. Tests have been done, for example, to figure out how many resources are needed to create a one-million-user Comet server.[‡] The key requirements to achieving this scale are straightforward in principle: utilize as few system resources as possible per connection, and write solid tests[§] to measure performance.

Servers must minimize the use of resources, while also optimizing the number of long-held connections based on the frequency and payload size of the amount of data to send to each client. Larger payloads and more frequent sending of data increase the latency and reduce the number of maximum possible connections of a Comet server.

Because Comet is really just a performance optimization of HTTP and connection management, performance measurement techniques are actually quite similar to those for measuring any large-scale web application.

Protocols

A Comet connection differs from the communication semantics applied over the connection. Comet connections allow either server→client communication only, or bidirectional communication. Various protocols are then layered on top of the connection to provide more functionality and better semantics than simply "read" and "write," such as Bayeux's Publish-Subscribe (PubSub) model.

Bayeux,[‖] is a protocol for transporting asynchronous messages (primarily over HTTP), with low latency between a web server, created as part of the cometD[#] project at the Dojo Foundation. Having a simple, extensible protocol is extremely beneficial for interoperability between various Comet servers and clients.

The PubSub paradigm is one approach commonly used with protocols such as Bayeux, with other protocols such as XMPP more common for chat applications.

The Dojo Toolkit provides all of the transport level handling of long polling and callback polling (including multiple frames for parallel JSONP requests) in its cometD module, and it also handles Bayeux service negotiation and communication. Conse-

[‡] *http://www.metabrew.com/article/a-million-user-comet-application-with-mochiweb-part-3/*

[§] *http://aleccolocco.blogspot.com/2008/10/gazillion-user-comet-server-with.html*

[‖] Bayeux is also the name of the tapestry showing the events leading up to the 1066 Norman invasion of England that includes Halley's Comet, which was believed to be a sign of impending doom.

[#] The *D* stands for daemon, much like the *d* in httpd.

quently, you can use Dojo with a Bayeux-compatible server simply by instantiating the cometD module and letting it handle the transport details:

```
dojox.cometd.init("/cometd");
dojox.cometd.subscribe("/some/topic", function(message){
    // callback function
});
```

Summary

Unlike Ajax, Comet is a complex set of performance optimization techniques that impact the client side, the server side, and communication between the two. It's still very early in the process of finding the right set of solutions for solving the Comet problem, but with the emergence of WebSocket, and solutions for handling millions of users, the complexity will decrease over time.

CHAPTER 9
Going Beyond Gzipping

Tony Gentilcore

Besides proper configuration of HTTP caching headers, enabling gzip compression is typically the most important technique for speeding up your web page. A chapter was devoted to compression in Steve Souders' first book, *High Performance Web Sites*. Now that all browsers support gzip and all responsible web developers have enabled gzipping, that chapter is closed, right? Not quite.

Even if you have enabled gzipping, there is a good chance that a small but significant portion of visitors to your site are not receiving compressed responses. The exact percentage varies greatly across different demographics and geographies, but a large web site in the United States should expect that roughly 15% of visitors don't indicate gzip compression support. This chapter explains why the percentage is higher than expected, how that affects performance, and what developers can do about it.

Why Does This Matter?

With such a small percentage, you might ask, "What's the big deal?" Let's take a look at what happens to 10 popular web sites when gzipping is disabled.

In this experiment, the page load time* for 10 popular web sites was measured by loading each web site 100 times in Internet Explorer 7.0 on Windows XP Pro. The cache remained primed between iterations to better represent the typical experience. All requests traveled through a proxy (Eric Lawrence's Fiddler†) on the same machine. In the control group the proxy did nothing, but in the experimental group it stripped the request's `Accept-Encoding` HTTP header so that compression was suppressed. Table 9-1 shows the absolute and percent increase when compression was disabled.

* Page load time was taken to be the time between the `OnBeforeNavigate2` and `OnDocumentComplete` events.

† *http://www.fiddlertool.com/*

Table 9-1. Page load time increase with compression disabled

Web site	Total download size increase (on first load)	Page load time increase (1000/384 Kbps DSL)	Page load time increase (56 Kbps modem)
http://www.google.com	10.3 KB (44%)	0.12s (12%)	1.3s (25%)
http://www.yahoo.com	331 KB (126%)	1.2s (64%)	9.4s (137%)
http://www.myspace.com	441 KB (143%)	8.7s (243%)	42s (326%)
http://www.youtube.com	236 KB (151%)	3.3s (56%)	21s (87%)
http://www.facebook.com	348 KB (175%)	9.4s (414%)	63s (524%)
http://www.live.com	41.9 KB (41%)	0.83s (53%)	9.2s (99%)
http://www.msn.com	195 KB (77%)	1.6s (32%)	13s (85%)
http://www.ebay.com	245 KB (92%)	1.7s (59%)	3.5s (67%)
http://en.wikipedia.org	125 KB (51%)	5.0s (146%)	21s (214%)
http://www.aol.com	715 KB (111%)	7.4s (47%)	32s (60%)
Average	**269 KB (109%)**	**3.9s (91%)**	**22s (140%)**

With compression disabled, in an empty cache state on the first load, the total size of all resources that had to be downloaded more than doubled. Note that this number does not indicate the gzip compression ratio because the total download size is taken to be the sum of all resources downloaded, including images and Flash. Gzip compression is generally applied only to textual resources such as HTML, CSS, and JavaScript files.

For DSL users, the average page load time increased from 4.3 to 8.3 seconds, a 91% increase. Dial-up users have it much worse, with an average page load time increase of from 15 to 37 seconds, a 140% increase.

With this data in hand, we can get back to the original question: "Should I care about the users who miss out on compression?" A naïve attempt at answering this question would be to calculate the average benefit across all requests: 15% of users times a 91% slowdown equals a 14% slowdown averaged over all requests. If your mean page load time is four seconds, that means on average it is slowing your users by only 560ms. You may not think anyone is going to leave your web site in that half of a second, so why care?

This is a case where looking at the averaged benefit does not tell the real story. In actuality, 85% of users are unaffected, but the 15% of users who are affected are affected in a big way. An additional four seconds is enough to cause users to abandon your web site. It is important to understand why content may not be compressed and whether there is anything developers can do.

What Causes This?

Now that you realize this is a real problem affecting real users, the next logical step is to figure out what is going on so that you might stand a chance of fixing it.

Quick Review

Let's start with a review of how compression works. All modern browsers (since the 4.*x* generation, circa 1998) support gzip compression and indicate that to web servers by supplying the `Accept-Encoding` HTTP header:

```
Accept-Encoding: gzip, deflate
```

When this header is present in the request and gzip compression is enabled on your web server, in compliance with RFC 2616 section 14.3‡ it responds with a compressed response marked by the `Content-Encoding` header:

```
Content-Encoding: gzip
```

The Culprit

If all modern browsers send the `Accept-Encoding` header, why are 15% of responses being served uncompressed? Surely, 15% of people aren't using browsers that are more than 10 years old. An analysis of a large sample of web server logs gave a clue to the culprit. Some requests arrived with mangled `Accept-Encoding` headers:

```
Accept-EncodXng: gzip, deflate
X-cept-Encoding: gzip, deflate
XXXXXXXXXXXXXX: XXXXXXXXXXXX
--------------: -------------
~~~~~~~~~~~~~~: ~~~~~~~~~~~~
```

But these mangled headers did not account for the full number of requests that are served without compression. Many more requests identified as real users in modern browsers (not bots) were simply missing the `Accept-Encoding` header altogether. Why would anyone or anything intentionally slow down users' web browsing experience by disabling compression? The culprits fall into two main categories: web proxies and PC security software.

What do these have in common? They both need to observe (or, if you prefer, spy on) the responses sent by the web server. Observing a response is cheaper in terms of CPU usage if the response does not have to be decompressed first. This, unfortunately, ignores the fact that, from the end user's perspective, the increased network time usually far outweighs the CPU time that would be necessary for the observing program to unzip the response.§ For this reason, I like to refer to the technique of stripping the

‡ *http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.3*

§ Especially considering that unzipping is typically three to four times faster than zipping.

`Accept-Encoding` header for the purpose of observing the responses as *turtle tapping*. Turtle tapping is the way a turtle might perform wire tapping: very slowly.

Examples of Popular Turtle Tappers

Table 9-2 shows several popular client software programs and web proxies along with how they modify the client's `Accept-Encoding` request header.‖ This list is by no means comprehensive. For instance, there are many add-ons for the popular Squid web proxy that strip or mangle the header to filter or observe web content.

Table 9-2. Software modifications to the Accept-Encoding header

Software	Accept-Encoding modification
Ad Muncher	Stripped
CA Internet Security Suite	`Accept-EncodXng: gzip, deflate`
CEQURUX	Stripped
Citrix Application Firewall	Stripped
ISA 2006	Stripped
McAfee Internet Security 6.0	`XXXXXXXXXXXXXX: ++++++++++++`
Norton Internet Security 2005	`--------------: ------------`
Novell iChain 2.3	Stripped
Novell Client Firewall	Stripped
WebWasher	Stripped
ZoneAlarm Pro 5.5	`XXXXXXXXXXXXXX: XXXXXXXXXXXX`

It is also interesting to examine how the percentage of turtle tapping victims varies with geography. The majority of requests coming from some Middle Eastern countries don't have a valid `Accept-Encoding` header. It is possible this could be due to a national firewall. Upward of 20% of users in the United States and Russia suffer from this problem. The European Union and Asian nations seem to have the best handle on this problem with fewer than 10% of users affected.

How to Help These Users?

Now that you have a good understanding of the problem's cause and effect, let's move on to the real work: helping these users get the fast experience they deserve (not to mention helping you get the benefit that happy users provide to your web site). Of course, the correct solution to this problem is to appeal to the vendors of software that strips or mangles the `Accept-Encoding` header. In fact, the problem is fixed in some

‖ Older or newer versions may not behave the same way, and it may be possible to change the program's behavior depending on the configuration.

newer versions. For instance, Norton Internet Security 2009 no longer causes this problem.

However, it will be some time before all existing users of these programs have switched or upgraded. Until then, this chapter discusses three approaches to mitigating this problem, each increasing in aggressiveness.

Design to Minimize Uncompressed Size

This may seem too obvious to bear repeating, but it cannot be stressed enough: *sending smaller responses makes pages faster*. This is why compressing responses is such an effective technique despite its CPU cost on both the client and the server. Good web developers know to do everything in their power to make HTML, CSS, and JavaScript as compact as possible. However, we have all been taught to think of long repeated strings as practically free because gzip compression makes them essentially disappear. As a result, we don't do much to optimize them. This assumption is completely invalid when considering users that cannot receive compressed responses.

Finding repetitive content that can be factored out is a bit of an art that varies highly across unique web sites. These are a few generally applicable techniques that can reduce your page's uncompressed size without increasing the compressed size.

Use event delegation

Often, several elements on a page require a similar event handler. Common examples that appear in most of the top 10 web sites are drop-down boxes, click-tracking links, and hover animations. The cost of specifying each handler in terms of page size adds up very quickly.#

For example, at the time of this writing, *http://facebook.com* includes a drop-down box with roughly 50 language options. Each language anchor in the drop-down requires an extra 133 uncompressed bytes to attach the `onclick` handler:

```
<a href="http://es-la.facebook.com/" onclick="return wait_for_load(this, event,
function() { intl_set_cookie_locale("", "es_LA"); return false;
});">Español</a>
```

Multiplying that waste times 50 links means about 6.7 KB of waste is being transferred to users that don't support compression. The distinct information repeated for each anchor is the URL, locale code, and language name.

Event delegation is the name commonly given to the technique of attaching a single event handler to a parent element that contains all of the elements that need to respond to the event. When the event is triggered on the child element, it bubbles up to the parent where it is handled. That single handler can distinguish which child element is

For interactive pages, reducing the number of event handlers can have an even more significant performance benefit in terms of JavaScript execution time.

the target of the event and receive additional parameters via some attribute on that element.

For example, to improve our *facebook.com* example, the links could be coded as follows:

```
<div class="menu_content" onclick="return intl_set_cookie_locale(event)">
  ...
  <a href="http://es-la.facebook.com/" class="es_LA">Español</a>
  ...
</div>
```

The new event delegation handler gets the locale, which was previously passed as a parameter, from the target element's class:

```
<script>
  function intl_set_cookie_locale(e) {
    e = e || window.event;  // Get event object.
    var targetElement = e.target || e.srcElement;  // Get target element.
    var newLocale = targetElement.class;  // Get the new locale.
    ...
    // Use newLocale to set the cookie.
    ...
    return false;  // Cancel the anchor's href action.
  }
</script>
```

The small amount of additional code to make the delegation work is insignificant because it can be placed in an external, cacheable JavaScript file instead of in the root document, which has to be downloaded every time the user visits the page.

Use relative URLs

We are all familiar with path-relative URLs (e.g., */index.html* instead of *http://www.example.com/index.html*). But as described by RFC 1808,[*] there are several lesser-known ways to make URLs relative that are supported by nearly all browsers going back to circa 1995. For example, with the notable exception of *slashdot.org*, almost no major web site employs protocol relative links (e.g., *//www.example.com* instead of *http://www.example.com*). Given the number of URLs on typical pages, the bloat of all those nonrelative URLs can quickly add up to a significant portion of the page size. Given a base URL of *http://www.example.com/path/page.html*, the relative URLs shown in Table 9-3 may be used.

Table 9-3. Relative equivalents of http://www.example.com/path/page.html

Fully specified destination URL	Relative equivalent
http://subdomain.example.com/	*//subdomain.example.com*
http://www.example.com/path/page2.html	*page2.html*

[*] *http://www.w3.org/Addressing/rfc1808.txt*

Fully specified destination URL	Relative equivalent
http://www.example.com/index.html	*/index.html*
http://www.example.com/path2/page.html	*../path2/page.html*
http://www.example.com/path/page.html#f=bar	*#f=bar*
http://www.example.com/path/page.html?q=foo	*?q=foo*

For dynamically generated URLs, it is trivial to write a function that will make each URL as relative as possible given the base URL of the page it includes.

Strip whitespace

Your users don't care how readable your code is, but they do care how fast your site is. Line breaks and proper indentation are invaluable to developers, but they should always be stripped by an automated process before being served to users. As discussed in *High Performance Web Sites*, many tools are available to do this for JavaScript. Some of the most popular are YUI Compressor, ShrinkSafe, and JSMin. For CSS, YUI Compressor does the best job. In HTML, the problem is a bit trickier because whitespace can be significant in many contexts. However, if you are willing to indicate the places where significant whitespace is needed, most major template languages have an option to strip leading and trailing whitespace as well as line breaks.

Strip attribute quotes

Before discussing stripping quotes around HTML attributes, two disclaimers need to be mentioned. First, if your web page is written in XHTML, attributes must be quoted. Second, HTML attributes should always be written with double quotes in place to avoid accidentally introducing bugs when attribute values change from a value that doesn't require quotes to one that does. However, according to the HTML 4.01 specification section 3.2.2,[†] it is valid to omit quotes around attributes that contain only letters, numbers, hyphens, periods, underscores, and colons (matching the regular expression `[a-zA-Z0-9\-._:]`).

To improve download time, it is beneficial to strip unnecessary quotes via an automated process before serving to the user.

Avoid inline styling

Another way in which the uncompressed page size is often unnecessarily inflated is by repeatedly styling content inline instead of relying on CSS. For example, at the time of this writing, *wikipedia.org* contains 4 KB of repeated inline style attributes throughout the main HTML document. This will gzip away quite efficiently, but it adds up to a significant amount of extra data to download when compression has been suppressed.

[†] *http://www.w3.org/TR/html4/intro/sgmltut.html#h-3.2.2*

Alias JavaScript names

Several commonly used DOM methods were given unfortunately long names in JavaScript. Compression usually makes repeating these names practically free. However, in the uncompressed case they can be quite expensive. Fortunately, JavaScript also allows us to overcome this problem by creating references (or aliases) to these long names.

The first place to look for aliasing opportunities is functions that are used frequently throughout your script. For instance, some popular JavaScript libraries alias `document.getElementById` as the variable `$`:

```
var $ = document.getElementById;
```

Throughout your script, you can then simply write `$("foo")` instead of writing `document.getElementById("foo")`. This saves 22 uncompressed bytes per use. It is usually wise to alias any method used more than three times.

The second place where aliases are beneficial is when accessing chained properties of an object.‡ This is best illustrated with an example:

```
// Wasteful
var foo = $("foo");
foo.style.left = "0";
foo.style.right = "0";
foo.style.height = "10px";
foo.style.width = "10px";

// Better
var foo = $("foo").style;
foo.left = "0";
foo.right = "0";
foo.height = "10px";
foo.width = "10px";
```

Real-world savings

How well do these techniques work? Table 9-4 shows the size reduction of the uncompressed root document achieved by applying each technique on the same set of popular web pages.§

Table 9-4. Size reduction achieved on popular web pages

Web site	Event delegation	Relative URLs	Strip space	Strip quotes	Use CSS	Total
http://www.google.com	1.8%	3.4%	--	--	0.4%	5.6%
http://www.yahoo.com	--	0.8%	3.3%	0.6%	0.5%	5.2%
http://www.myspace.com	4.0%	2.2%	9.0%	1.5%	1.8%	18.5%

‡ Aliasing in tight loops can also significantly improve JavaScript execution performance.

§ JavaScript aliasing was not feasible to test.

Web site	Event delegation	Relative URLs	Strip space	Strip quotes	Use CSS	Total
http://www.youtube.com	8.3%	0.6%	7.1%	2.3%	1.2%	19.5%
http://www.facebook.com	12.9%	1.7%	1.1%	2.6%	0.3%	18.6%
http://www.live.com	8.5%	0.9%	0.2%	0.9%	0.3%	10.8%
http://www.msn.com	--	3.0%	0.1%	1.7%	--	4.8%
http://www.ebay.com	0.2%	1.7%	1.2%	1.6%	1.2%	5.9%
http://en.wikipedia.org	--	1.6%	2.1%	1.8%	5.2%	10.7%
http://www.aol.com	10.4%	2.4%	1.4%	1.8%	0.5%	16.5%
Average	**4.6%**	**2.8%**	**2.6%**	**1.5%**	**1.1%**	**11.6%**

Applying all of these techniques trimmed the uncompressed size by anywhere from 5% to 20% (with an average of 11.6%). These optimizations are certainly worthwhile given that relative URLs, stripping whitespace, and stripping attribute quotes can be performed via an automated process. Furthermore, event delegation, JavaScript aliasing, and avoiding inline styles lead to more maintainable code that is also faster in the browser.

However, an 11.6% reduction simply pales in comparison to applying gzip, which reduces the same root documents by an average of 72.1%. Going back to the original timing tests, if applying gzip reduces page size by 72.1% and 3.9 seconds, we might expect that a reduction of 11.6% would lead to a speed gain of about 630ms.

To beat this problem, we are going to have to look beyond such workarounds and toward a method of reducing the number of responses that are served without compression.

Educate Users

Once your page's uncompressed size is as small as possible, another potential solution to help the victims of turtle tapping is to inform them of the problem. There is ample precedence for this type of helpful message on the Web. For users of the Firebug Firefox extension, Gmail displays a bright red dialog at the top of the page that reads, "Firebug is known to make Gmail slow unless it is configured correctly." Surf the Web for a while with Internet Explorer 6.0 and you'll notice messages prompting you to upgrade your browser.

This technique can be applied if the request does not contain a valid `Accept-Encoding` header. Displaying a brief, informative message such as the following could help your user correct the problem:

```
Your Internet connection is slowed because it does not allow compression.
                        Fix this                Hide
```

The "Fix this" link points to a page that explains the types of software that cause this and how to disable or upgrade them. The "Hide" link sets a cookie so that the message is never displayed again.

Unfortunately, this too is not an adequate solution. Users behind a proxy that is preventing compression are powerless to change anything beyond perhaps complaining to an administrator. There is one more strategy, described next, that can be used to help these users.

Direct Detection of Gzip Support

After all else has failed, if uncompressed responses are still causing pain for your site, there is one guerrilla tactic that may be considered: to directly test for compression support rather than relying on the `Accept-Encoding` header. This may sound dangerous initially, but if properly tested, it can be safe. It is important to get this right because you don't want to risk a single false positive. Expect that direct detection will allow you to compress roughly half of the requests that are missing compression.

Performing the test

If the `Accept-Encoding` header is missing from the request, conditionally output a hidden iframe as the last element of the page `<body>`:

```
<iframe src="/test_gzip.html" style="display:none"></iframe>
```

This will load a *test_gzip.html* document, which you set up as follows:

1. Disable caching so that the current connection is always tested.
2. Compress the contents, regardless of the request headers.
3. Use JavaScript to set a session-only cookie indicating that the browser supports gzip.

If the client supports compression, a cookie indicating that fact will be sent with subsequent requests. If the client truly does not support compression, the hidden iframe will just load garbled text that won't be seen and won't set the `supports_gzip` cookie.

There are many ways to accomplish this. Here is an example written in PHP:

```
<?php
  function flush_gzip() {
    $contents = ob_get_contents();
    ob_end_clean();
    header('Content-Type: text/html');
    header('Content-Encoding: gzip');
    header('Cache-Control: no-cache');
    header('Expires: -1');
    print("\x1f\x8b\x08\x00\x00\x00\x00\x00");
    $size = strlen($contents);
    $contents = gzcompress($contents, 9);
    $contents = substr($contents, 0, $size);
```

```
    print($contents);
  }

  ob_start();
  ob_implicit_flush(0);
?>

<html>
  <body>
    <script>
      document.cookie="supports_gzip=1";
    </script>
  </body>
</html>

<?php
  flush_gzip();
?>
```

Using the result

Now your subsequent web pages of the same `Content-Type` can be compressed if the `supports_gzip` cookie exists. When forcing compression based on the presence of the cookie, make sure that the response is not publicly cacheable, and don't bother to output the `test_gzip` iframe again.

Again, the implementation will vary based on your environment. Here is a PHP example that uses the same `flush_gzip` method defined previously:

```
<?php
  // flush_gzip() definition omitted for brevity.

  ob_start();
  ob_implicit_flush(0);
?>

<html>
  <!-- Your page goes here. -->
</html>

<?php
  if (isset($_COOKIE["supports_gzip"])) {
    flush_gzip();
  } else {
    flush();
  }
?>
```

Measuring the effectiveness

Always keep track of two important statistics when considering or using direct detection of gzip support. The first is the percentage of requests that don't indicate compression support via the `Accept-Encoding` header. If that percentage is too low, the

technique of directly detecting compression support is not worth the hassle. The second is the percentage of requests that are missing the `Accept-Encoding` header but are found to support compression. This can be measured only after direct detection. Direct detection should continue to be employed only if this percentage remains high.

CHAPTER 10

Optimizing Images

Stoyan Stefanov and Nicole Sullivan

The single most important thing you can do to improve performance is put your site on a diet—take off (and keep off) all the bytes you put on under the stress of chasing the next killer feature. Optimizing images is one way to do just that. Historically, the question of which features to include was considered a business rather than an engineering decision, so page weight has rarely been discussed in performance circles, and yet it is extremely important to overall response time.

Response time for web pages is almost exactly correlated to page weight, and images tend to account for half of the size of typical web pages (see Figure 10-1). Most importantly, images are an easy place to improve performance without removing features. Often, we can make substantial improvements in the size of an image with little to no reduction in quality.

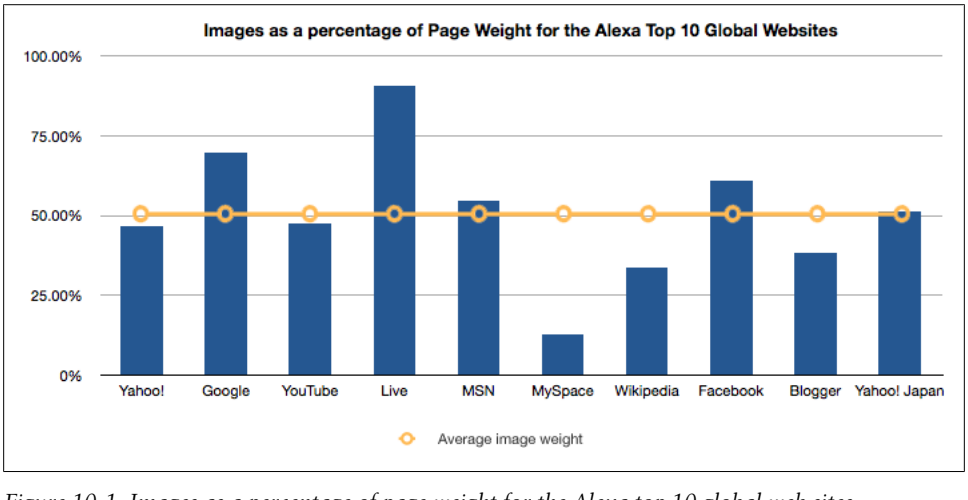


Figure 10-1. Images as a percentage of page weight for the Alexa top 10 global web sites

In this chapter, we focus on *nonlossy* optimizations, which result in a smaller overall file size with no loss in quality. Pixel for pixel, the visual quality of the original and final images is the same. The reduction in size often results from removing metadata, better compression of color or pixel information, or (in the case of PNG) removing chunks that are not necessary for the Web.

If you don't optimize images, you send extra data over the wire that adds nothing to the user experience. It seems like a no-brainer to follow the practices we'll recommend in this chapter, but image optimization falls in the blurry space between engineering and design, and has historically been a neglected part of the performance puzzle.

In this chapter, we'll cover:

- Characteristics of different image formats for the Web (GIF, JPEG, and PNG)
- Automating lossless optimization
- The `AlphaImageLoader` filter
- Optimizing sprites
- Other image optimizations

Two Steps to Simplify Image Optimization

Image optimization is simpler when it is broken down into two steps, each of which is owned by a different stakeholder in the creation of a web site:

1. Optimizing images begins with a qualitative decision about the number of colors, resolution, or accuracy required for a given image. These changes are *lossy optimizations* that result in an overall loss of quality. The image might have fewer colors, or in the case of the JPEG format, less detailed encoding. Although 60% to 70% quality is the accepted standard for JPEG, some images or contexts may require more or less quality. For instance, glossy images of celebrities may require a larger file size than autogenerated charts or tiny thumbnails. These decisions are creative decisions and should be made by the designer, using tools such as the Save for the Web feature in Photoshop. The designer may also choose to do "spatial" or "zonal" compression—for example, choosing 80% quality for Brangelina's face and only 30% quality for the night background.
2. Once the quality choice has been made, use *nonlossy compression* to squeak the last bytes out of the image. Unlike the preceding step, this one begs for an engineering solution. Doing the same work by hand would be much more time-consuming. In fact, fantastic open source tools exist for optimizing images. You can write a script that goes over all of your image files, determines the type of each, and runs a utility to optimize the file.

Image Formats

The first step in producing optimal images is to understand the features of each of the three formats used on the Web today—JPEG, PNG, and GIF—and choosing the right one for each specific case. Let's start the discussion of the different formats with just a few bits of background information.

Background

This section discusses the traits of images that affect how you use them on the Web and that factor into your choice of a format.

Graphics versus photos

Both the image format you use and the ways to optimize it depend on which of the following categories the image falls into:

Graphics
: Examples of graphics are logos, diagrams, graphs, most cartoons, and icons. These images usually contain continuous lines or other sharp transitions in color. The number of distinct colors in a graphic is relatively small.

Photos
: Photos usually have millions of colors and contain smooth color transitions and gradients. Imagine, for example, a picture of a sunset you take with your camera. An image of a painting (such as the *Mona Lisa*) is also closer to a photo than a graphic.

In terms of formats, GIFs are often used for graphics, whereas JPEG is the preferred format for photos. PNG comes in two kinds, of which *palette PNG* is even better suited for graphics than GIF.

Pixels and RGB

Images consist of *pixels*, where a pixel is the smallest piece of image information. Different color models can be used to describe a pixel, but the RGB color model is the one usually used for computer graphics.

In the RGB color model, a pixel is described based on the amount of red (R), green (G), and blue (B) it contains. R, G, and B are called *components* (a.k.a. *channels*), and the intensity of each component has a value from 0 to 255. The hexadecimal representation of the channel values, often used in HTML and CSS, ranges from 00 to FF. Mixing different intensities of the three channels gives you different colors. For example:

- Red is `rgb(255, 0, 0)` or hex `#FF0000`.
- Blue is `rgb(0, 0, 255)` or hex `#0000FF`.

- A shade of gray will likely have equal parts of each color; for example, `rgb(238, 238, 238)` or hex `#EEEEEE`.

Truecolor versus palette image formats

Using the RGB color model, how many distinct colors can you represent in a graphic? The answer is more than 16 million: 255 * 255 * 255 (or 2^{24}) gives you 16,777,216 combinations. Image formats that can represent this many colors are called *truecolor image formats*; examples are JPEG and the truecolor type of PNG.

To save space when storing the image information in a file, one technique is to create a list of all the unique colors found in the image. The list of colors is called a *palette* (also called an *index*). Having the list of colors, you can represent the image by keeping track of which palette entry corresponds to each pixel.

The palette can contain any RGB value, but the most common palette image formats—GIF and PNG8—limit the number of palette entries to 256. This doesn't mean you can pick from only 256 predefined colors. On the contrary, any of the 16+ million colors are up for grabs, but you can only have up to 256 of them in a single image.

Transparency and alpha channel (RGBA)

RGBA is not a distinct color model, but more of an extension to RGB. The extra component *A* represents alpha transparency and also has values from 0 to 255, although different programs and libraries define it as a percentage from 0% to 100% or values from 0 to 127. The alpha channel describes how much you can see through the image pixel.

Let's say you have a web page that has a background pattern and a blue image on top of it. If a pixel in the image has zero alpha transparency, the background behind the image will not be visible. If the alpha transparency value is the maximum 100%, the pixel will not be visible at all and the background will "shine through." A medium value of, say, 50% will let you see both the background and the pixel. Figure 10-2 shows some examples.

Interlacing

When a large image downloads over a slow Internet connection, it is drawn as it arrives, one row at a time from top to bottom, so it grows down slowly. To improve the user experience, some image formats support *interlacing*, in which successive samples of the image are shown. Interlacing lets the user see a rough version of the image while waiting for the details, giving the psychological effect of eliminating the feeling that the page is delayed.

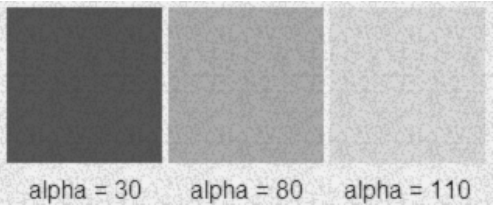


Figure 10-2. Examples of images with variable transparency produced using PHP with the GD library, which declares alpha values from 0 to 127

Characteristics of the Different Formats

With this background under our belts, let's see how GIF, JPEG, and PNG differ.

GIF

GIF, an abbreviation for Graphics Interchange Format, is a palette image format. Here are some of its features:

Transparency
: GIF allows for a binary (yes/no) type of transparency—a pixel is either fully transparent (not containing a color) or fully opaque (containing a solid color). This means that alpha (variable) transparency is not supported; instead, one of the colors in the palette is marked to represent transparency, and transparent pixels are assigned that color. So, if your GIF has transparent pixels, this will "cost" you one palette entry.

Animation
: The GIF format supports animation. An animated image consists of a number of frames; it's like having several images contained in the same file. GIF animations are generally perceived as annoying because of their abuse in the early years of the Web, when they were used for blinking text, rotating @ signs, and so on. They still have some application today; for example, for ad banners (although this is mainly a Flash domain now) or little "Loading..." indicators in Rich Internet Applications (RIAs).

Nonlossy
: The GIF format is nonlossy, which means you can open a GIF, do some editing, and save it without losing quality.

Horizontal scanning
: When writing a GIF file, a compression algorithm (called *LZW*) is used to reduce the file size. When compressing the GIF, the pixels are scanned horizontally, top to bottom. This results in a better compression when you have areas of horizontally repeating colors. For example, a 500 × 10-pixel image (width: 500px; height: 10px) containing stripes—meaning horizontal lines of the same color—will have a

smaller file size than the same image rotated to 90 degrees (width: 10px; height: 500px) when the stripes become vertical.

Interlacing
: GIF supports optional interlacing.

The 256-color limit for GIFs makes them unsuitable for photos, which usually require a much greater number of colors. GIFs are better suited for graphics (icons, logos, diagrams), but as you'll see later in this chapter, PNG8 is a superior format for graphics. Therefore, you should usually use GIFs only for animation.

There used to be a patent issue with LZW, the lossless data compression algorithm used by the GIF format, but the patents expired in 2004, so GIF can be used freely now.

JPEG

JPEG stands for Joint Photographic Experts Group, the organization that developed the standard. JPEG is the de facto standard for storing photos. This format reduces the information required to show a picture through techniques that take into account the human eye's perception of color and light intensities, so it can store high-resolution images in greatly compressed files. Here are some of its features:

Lossy
: JPEG is a lossy format that accepts a user-specified quality setting, which determines how much image information is lost. The quality values range from 0 to 100, but even a value of 100 will result in some quality loss.

  When you do multiple edits of the same image, it's best to use a nonlossy format to store the intermediate results and then save as JPEG once you're done with the changes. Otherwise, you'll lose some quality every time you save.

  A few operations can be performed losslessly, such as:

  - Rotation (only to 90, 180, or 270 degrees)
  - Cropping
  - Flipping (horizontal or vertical)
  - Switching from baseline to progressive and vice versa
  - Editing image metadata

  The last of these operations is particularly valuable for our purposes. We'll exploit it later to automate the optimization of JPEGs.

Transparency and animation
: JPEG doesn't support transparency or animation.

Interlacing
: In addition to the default baseline JPEG, there's also a progressive JPEG, which supports interlacing. Internet Explorer doesn't render the progressive JPEG in stages, but it successfully displays the whole image once it arrives.

JPEG is the best format for photographic images on the Web and is also widely used in digital cameras. It is not suitable for graphics, however, because of the artifacts of the lossy compression around lines or other sharp transitions of color.

PNG

PNG (Portable Network Graphics) was created to address shortcomings of the GIF format and its patent complications. In fact, the joke goes that PNG is a recursive acronym that stands for "*P*NG is *N*ot *G*IF." Here are some of its features:

Truecolor and palette PNGs
: The PNG format has several subtypes, but they can roughly be divided into two: palette PNGs and truecolor PNGs. You can use palette PNGs as replacements for GIFs, and you can use truecolor PNGs instead of JPEGs.

Transparency
: PNG supports full alpha transparency, although there are two quirks in Internet Explorer version 6 that we'll describe later.

Animation
: Although experiments and actual implementations exist, currently there's no cross-browser support for animated PNGs.

Nonlossy
: Unlike JPEG, PNG is a nonlossy format: multiple edits do not degrade quality. This makes the truecolor PNG a suitable format for storing intermediate stages of editing a JPEG.

Horizontal scanning
: Like GIFs, PNGs that have areas of horizontally repeating colors will compress better than those with vertically repeating colors.

Interlacing
: PNG supports interlacing and uses an algorithm that is superior to GIF; it allows for a better "preview" of the actual image, but interlaced PNGs have bigger file sizes.

More About PNG

Let's take a look at a few more details that will give you a better understanding of the PNG format.

PNG8, PNG24, and PNG32

You might come across the names PNG8, PNG24, or PNG32. Let's clarify their meaning:

PNG8
: Another name for palette PNG

PNG24
: Another name for truecolor PNG that has no alpha channel

PNG32
: Another name for truecolor PNG with alpha channel

There are other variations, such as grayscale PNGs with and without alpha, but they are used much more rarely.

Comparing PNG to the other formats

It's clear that GIFs are designed for graphics, JPEGs for photographs, and various types of PNGs for both. This section compares PNG to the other formats and offers some extra details about PNG.

Comparison to GIF

Except for animation support, palette PNGs have all the features of GIFs. In addition, they support alpha transparency and generally compress better, resulting in smaller file sizes. So, whenever possible, *you should use PNG8 rather than GIF*.

One exception is that very small images with very few colors might compress better as GIFs. But such small imagery should be part of a CSS sprite, because the "price" of an HTTP request will greatly outweigh the saving of a few bits. Chances are the sprite image will compress better as a PNG.

Comparison to JPEG

When you have an image with more than 256 colors, you need a truecolor image format—a truecolor PNG or a JPEG. JPEGs compress better and, in general, *JPEG is the format for photos*. But since JPEGs are lossy and there are artifacts around sharp transitions of color, there are cases when a PNG is better:

- When the image has slightly more than 256 colors, you might be able to convert the image to PNG8 without any visible quality loss. It's quite surprising how sometimes you can strip out more than 1,000 colors and still not notice the difference.
- When artifacts are unacceptable—for example, a color-rich graphic or a screenshot of a software menu—a PNG is the preferred choice.

PNG transparency quirks

Two quirks in Internet Explorer 6 are related to PNG and transparency:

- Any semitransparent pixels in a *palette PNG* appear as fully transparent in Internet Explorer 6.
- Alpha transparent pixels in a *truecolor PNG* appear as a background color (most often gray).

The first issue means PNG8 behaves like GIF in Internet Explorer 6. This is not so bad and still allows you to select PNG instead of GIF for all your graphical images. PNG8, therefore, offers "progressively enhanced" semitransparent images that look great in all modern browsers and degrade to GIF-like transparency in Internet Explorer 6.

The second issue is a little more serious and there are various workarounds that boil down to the use of the proprietary CSS property `AlphaImageLoader` or the use of VML. As you'll see later in this chapter, `AlphaImageLoader` comes at a cost in performance and user experience and you should avoid it when possible. The VML workaround has the drawback of adding extra markup and code. In conclusion, always try to achieve the design using PNG8.

PNG8 and image editing software

Unfortunately, most image-editing programs, including Photoshop, can only save PNG8 with binary transparency. One notable exception is Adobe Fireworks, which has excellent alpha transparency support. There are also command-line tools such as pngquant and pngnq that allow you to convert truecolor PNGs to palette PNGs.

Here's an example of a pngquant command, where the number 256 specifies the maximum number of colors in the palette:

```
pngquant 256 source.png
```

Automated Lossless Image Optimization

Now that you know about the different image formats, let's see how you can optimize your images. The beauty of the process you're about to see is that:

- It's automated and doesn't require human interaction.
- All operations are lossless, so you don't have to worry that the image quality will degrade.
- It uses freely available command-line tools.

Each image type requires different handling, but it's usually predictable and easy to automate in a script. This section discusses the following tasks:

- Crushing PNGs
- Stripping JPEG metadata
- Converting single-image (nonanimated) GIFs to PNGs
- Optimizing GIF animations

Crushing PNGs

PNGs store image information in "chunks." This makes the format extensible because you can add more functionality to it using custom chunks, and programs that do not

understand your new extensions can safely ignore them. But most of the chunks are not needed for web display, and you can safely remove them. An additional benefit is that stripping the so-called gamma chunk actually improves the cross-browser visual results, because each browser treats gamma corrections slightly differently.

Pngcrush

Our favorite tool for PNG optimization is pngcrush. You can run it like this:

```
pngcrush -rem alla -brute -reduce src.png dest.png
```

Let's take a look at the options:

`-rem alla`
: Removes all chunks except the one controlling transparency (alpha).

`-brute`
: Tries more than 100 different methods for optimization in addition to the default 10. It's slower and most of the time doesn't improve much. But if you're doing this process offline, you can afford the one or two more seconds this option takes, in case it finds a way to cut the image size further. Remove this option in performance-sensitive scenarios.

`-reduce`
: Tries to reduce the number of colors in the palette, if possible.

`src.png`
: The source image.

`dest.png`
: The destination (result) image.

Other PNG optimization tools

Pngcrush hits a pretty good middle ground that balances execution speed against optimization results. But if you want to achieve the best possible results and you're prepared to spend a little more time on optimization, you can try some of the other tools. Results vary, depending on the image. You can even run all the tools in succession.

Notable tools include:

PNGOUT
: Binary-only, Windows, closed source

OptiPNG
: Cross-platform, open source, command-line interface

PngOptimizer
: Windows, open source, GUI and command-line interfaces

One “heavy-duty” tool is also available: PNGslim. It’s a batch file for Windows that runs a number of other tools. Its main activity is to run PNGOUT hundreds of times with different options. PNGOUT is the slowest of all the tools we’ve tried, so you should be prepared to allow PNGslim plenty of time to run—sometimes hours to optimize a single file.

Stripping JPEG Metadata

JPEG files contain metadata such as the following:

- Comments
- Application-specific (e.g., Photoshop) internal information
- EXIF information such as camera make and model, the date the photo was taken, the geolocation of the photo, thumbnails, or even audio

This metadata is not used for image display and can safely be removed. Metadata handling, luckily, is one of the lossless JPEG operations mentioned earlier in this chapter, so you can remove the unneeded parts of the file without losing visual quality.

A tool called jpegtran does the transformation on the command line:

```
jpegtran -copy none -optimize src.jpg > dest.jpg
```

The options in this example are:

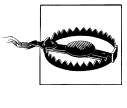
`-copy none`
: Instructs that no meta information should be carried over

`-optimize`
: Causes jpegtran to optimize the Huffman tables used for compression

`src.jpg`
: Your image before optimization

`dest.jpg`
: The optimized file

The command writes to standard output, so to create the final file, this example just redirects output to a file named *dest.jpg*.



Strip meta information only from images you own. By stripping metadata from someone else’s JPEG, you might also strip any copyright or authorship data, which is illegal.

Jpegtran takes an all-or-nothing approach to handling metadata. For more fine-grained metadata editing, use ExifTool.

Converting GIF to PNG

As we discussed, the PNG8 format supports everything that GIF does, so converting a GIF to PNG8 should result in no visible changes. You can use ImageMagick to do the conversion from the command line as simply as:

```
convert source.gif destination.png
```

You can also force the PNG8 format by using:

```
convert source.gif PNG8:destination.png
```

This is probably not necessary, since GIFs are likely to be converted to PNG8 anyway. ImageMagick picks the appropriate format based on the number of colors.

Once you've converted the GIF to PNG, *don't forget to crush the PNG result* (as shown earlier in this chapter).

You can also use ImageMagick's *identify* utility to programmatically determine whether the GIF file contains an animation. For example:

```
identify -format %m my.gif
```

This command will simply return "GIF" for nonanimated GIFs. For GIF animations it will return a string such as "GIFGIFGIF..." repeating "GIF" once for every frame. If you're running a script to convert files, checking for the presence of "GIFGIF" in the first six-character substring of the output will let you know you're dealing with an animated file. In that case, you can skip to the next step.

Optimizing GIF Animations

Now that all your single-image GIFs are PNGs, your PNGs are crushed, and your JPEGs are optimized, the last things left to optimize are the GIF animations. One tool that can help you is Gifsicle. Since the animations consist of frames and some parts of the image don't change from one frame to another, Gifsicle optimizes animations by removing the duplicate pixel information from successive frames. The way to run it is:

```
gifsicle -O2 src.gif > dest.gif
```

Smush.it

Smush.it is an online tool for image optimization, created by the authors of this chapter. It does what we just described in the previous four sections, applying lossless image compression to a variety of file types. Smush.it has a convenient Firefox extension companion that allows you to visit any page and optimize all the images on that page in one shot. You can always check how much you can save by following these steps.

Do note that Smush.it underperforms with JPEGs because it doesn't strip the JPEG metadata, since we don't want to involuntarily strip copyright information and "orphan" a JPEG. If you roll out your own "smushing" tool using the techniques and

tools described earlier and you're sure it's appropriate to remove the metadata, do so using jpegtran's `-copy none` option.

Progressive JPEGs for Large Images

When reviewing the different file formats, we mentioned that there are *progressive* JPEGs that render progressively in the browser, allowing the user to see a low-resolution version of the image while the file is still being transferred. The question is whether progressive JPEGs are smaller or bigger than nonprogressive equivalents.

After experimenting with more than 10,000 images chosen at random from the Web using the Yahoo! image search API, we've reached the conclusion that you cannot tell for sure. In fact, results can be all over the map. But a trend did emerge: images bigger than 10 KB usually compress better as progressive JPEGs. Smaller images are better as nonprogressive, *baseline* JPEGs. Figure 10-3 summarizes our findings, charting the original file size against the difference caused by optimization. The graphic ends at 30 KB, but the trend remains flat, meaning that the benefit of progressive encoding increases with file size.

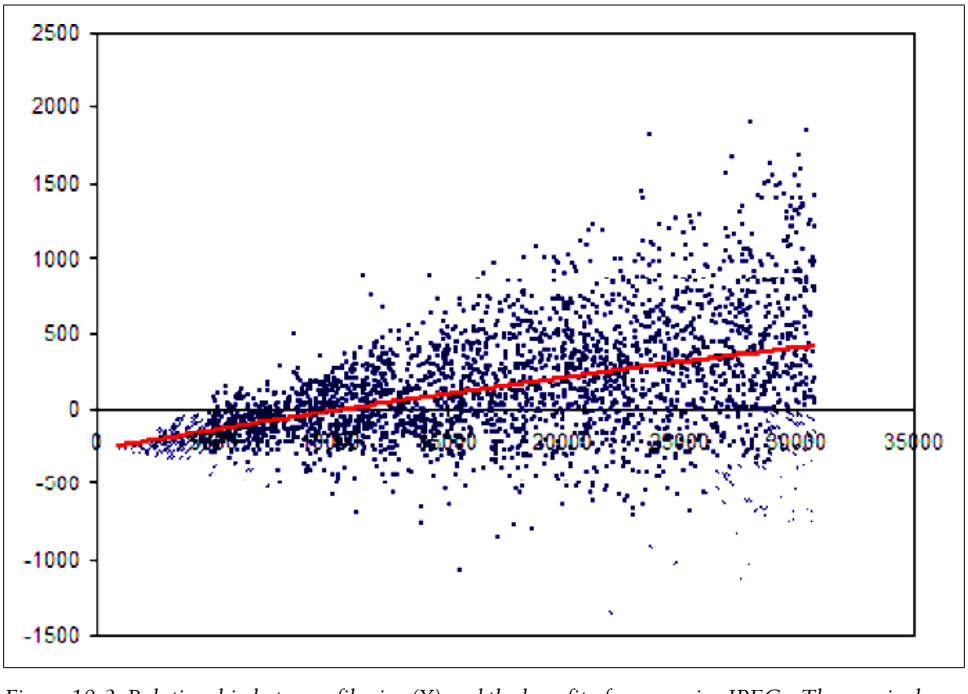


Figure 10-3. Relationship between file size (X) and the benefit of progressive JPEGs. The y-axis shows the difference between the file size of the baseline image and the progressive image; the greater the Y value, the better it is to use progressive encoding

Alpha Transparency: Avoid AlphaImageLoader

Much sought after in the world of web design, cross-browser alpha transparency is harder to achieve than you might expect. The PNG specification was written more than a decade ago, but lousy browser implementation means we're still looking for the one perfect solution. Support for truecolor PNG has evolved very slowly. Internet Explorer 6 still has significant market share, but it suffers from serious technical limitations in handling PNG alpha transparency.

In this section, we'll take a closer look at *alpha filters*, which force support for alpha transparency in older versions of Internet Explorer. In particular, Internet Explorer offers a filter called `AlphaImageLoader` that has become quite popular. Drawing on experimentation and practice at Yahoo!, and backed up by hard data, we've concluded that you should not use `AlphaImageLoader` to fix Internet Explorer 6 transparency problems. We'll explain why this is so and show practical examples of progressively enhanced PNG8 to work around these limitations.

Effects of Alpha Transparency

As you saw earlier, transparency comes in two flavors. The first is a kind of binary transparency: each pixel is either fully transparent or fully opaque. The second is alpha transparency, which allows you to have variable levels of opacity.

The lack of support for true alpha transparency in Internet Explorer 6 has been a challenge for web developers who wanted to have smooth transitions and drop shadows. For instance, the lefthand side of Figure 10-4 shows an effect we'd like to achieve (partial transparency that allows some of the background to show through); the righthand side shows the inclusion of the background color (in this case, white), which we have to live with when only binary transparency is supported. Solid-color backgrounds work equally well for both image formats, as they allow the binary transparent image to perfectly mimic the effect of full alpha transparency. However, the same icon could not be reused if the background color was not identical.

Figure 10-4. Alpha and binary transparency in My Yahoo! weather icons; the squares are a typical pattern used in image editing programs to denote transparency

Typically, alpha transparency is used in cases where the background is variable, as in a photograph, graphic, or gradient. In these cases, it is harder to fall back on simulating

transparency because it is impossible to be certain which colors will be behind a given portion of the image.

Perfect alpha transparency allows the image of the cloud to be put over any background and it will display beautifully. Binary transparency (on the right side of Figure 10-4) requires a slightly more creative approach: the designer is forced to approximate transparency by including a small portion of the background color around the graphic.

Gradient backgrounds (see Figure 10-5) require more careful attention to the edges of the cloud overlay. If too much of the fluffy edge or drop shadow is left around the image, it will look too dark in some areas and too light in others. It helps to leave as little background color as possible. In Figure 10-6, it's obvious that too much color was kept around the graphic. The mid-tone is too light at the top of the image and too dark at the bottom.

Figure 10-5. Gradients and transparency

Figure 10-6. A real-world example that shows the weather icon above a variable background on the My Yahoo! page; the design has all of the challenges discussed: gradients, solid colors, and patterns

Mountaintop corners

Another example is the ubiquitous rounded corner module. It is important to avoid including the background color of the module and the contour in one image because combining them will *drastically increase HTTP requests*. Rather than having one image

for the contour that can be combined with multiple backgrounds, you will have many images representing all the possible combinations.

Separating contour and page background colors from the block background color or image allows for scalable CSS. However, it is interesting to note that separating the two requires a very careful selection of pixels and reliance on the eye's tendency to smooth transitions. Dan Cederholm first wrote about this in his article "Mountaintop Corners" for A List Apart.

The difference between the two dark blocks in the module in Figure 10-7 may seem subtle, but when there is a high contrast between module, background, and foreground colors, the edge can seem "chewed off" rather than completely smooth. To achieve a uniform look and feel across browsers, developers began to use the `AlphaImageLoader` filter for Internet Explorer.

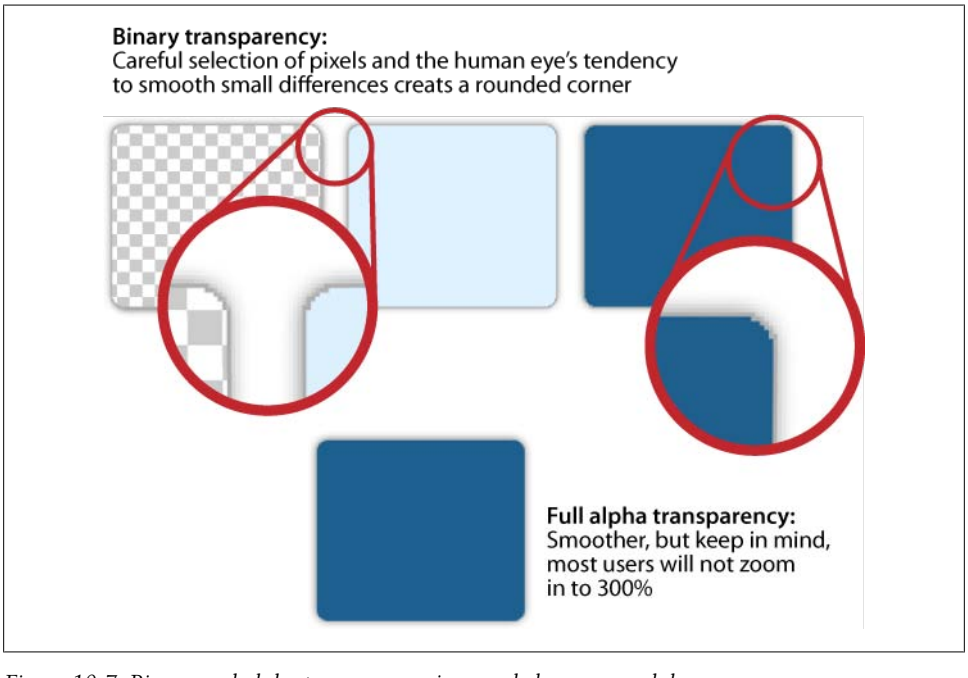


Figure 10-7. Binary and alpha transparency in rounded corner modules

AlphaImageLoader

Internet Explorer does not natively support alpha transparency. Proprietary filters are used to fill the gap; however, the performance costs of this choice are significant. To understand the drawbacks of the method, let's look at what not to do (see Example 10-1).

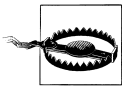
Example 10-1. Using AlphaImageLoader to add rounded corners to PNGs

```
.myModule .corner{
  background-image: url(corner.png);
  _background-image: none;
  _filter:progid:DXImageTransform.Microsoft.AlphaImageLoader(
     src='corner.png',
     sizingMethod='scale'
  );
}
```

In Example 10-1, the initial underscores are a hack that causes the attributes to be applied only to versions of Internet Explorer older than version 7:

- The `_background-image` attribute removes the original background, *corner.png*.
- The `_filter` attribute reloads the same image using Microsoft's `AlphaImageLoader` filter.

Only Internet Explorer 6 (or earlier) requires this hack. Internet Explorer 7 and later support alpha transparency natively, as do Firefox, Safari, and Opera. All these browsers ignore the rules with initial underscores, because the properties simply aren't recognized.



If you forget to use the underscore hack, Internet Explorer 7 will use the filter despite having native support for alpha transparency.

Problems with AlphaImageLoader

There are both maintenance costs and direct performance costs associated with using alpha filters:

Code forking
: Even the minimal amount of code forking in the prior example is dangerous from a maintenance point of view. When we write exceptions to CSS rules, the files tend to grow over time. Also, assuming we were using a CSS sprite to reduce HTTP requests, as recommended in *High Performance Web Sites*, `background-position` would not be supported *when the alpha filter is used*. In this case, the `clip` property is often used to simulate background image positioning in Internet Explorer.

Freezing the browser
: When an alpha filter is applied, the page does not render progressively. The user will see a blank page until all the necessary components are downloaded. Page elements can still be downloaded in parallel, but the display will be blocked because Internet Explorer will not render anything until every last bit of CSS comes down the wire, and the CSS has a dependency on a filtered image. (For more information about rendering, see *http://www.phpied.com/rendering-styles*.) If you have several `AlphaImageLoader` filters on the page, they are processed serially, multiplying the

problem. If you have five images, each delayed 2 seconds on the server, the browser freezes for a total of 10 seconds.

Increased memory consumption
: Another negative effect of using `AlphaImageLoader` is the increase in memory that is required to process and apply the filters. These days we might be tempted to think our visitors' computers have a virtually inexhaustible supply of memory, but that might not be the case for older computers, which are the ones more likely to run Internet Explorer 6 and earlier.

All this overhead applies to the image every time it appears on the page because the filter doesn't change the image, but rather the HTML element to which the style is applied. If a sprite is used on 20 HTML elements on the page, there isn't one performance penalty, but 20! Furthermore, each element is processed synchronously in a single UI thread. Often, this filter is used for "play" buttons that overlay video (see Figure 10-8). In this case, any performance penalty will be assessed for each video on the page.

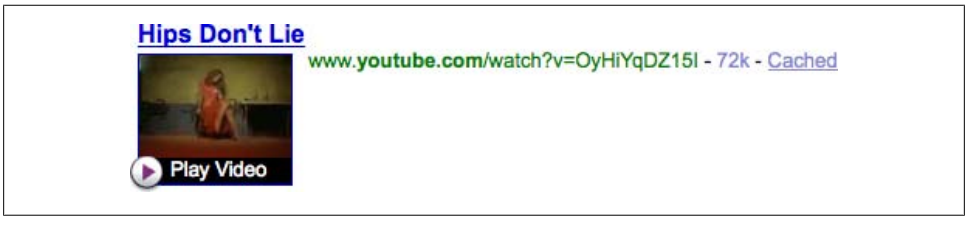


Figure 10-8. Video player at Yahoo! Search that had the alpha transparency from the "play" button removed, to improve performance

In the wild: A Yahoo! Search case study

Lab tests are an excellent way to estimate the performance impact of code changes, but there's nothing like testing an idea in the wild, with millions of requests coming from real users and their myriad browser configurations, geographic locations, connection speeds, hardware, and operating systems.

Based on lab tests, we estimated the performance "price" of the `AlphaImageLoader` filter to be approximately eight milliseconds per HTML element on which the filter is applied. Previously, the search team used a truecolor PNG and filter for their main sprite, which appeared 12 times in the page. Therefore, we expected approximately a 96-millisecond improvement, but we were eager to see whether a real user test would replicate our results.

The experiment compared two identical search results pages, one with the `AlphaImageLoader` filter and the other without. The results set comprised two distinct populations, which showed a 50- to 100-millisecond improvement. The response time for users of Internet Explorer 6 showed a 100-millisecond improvement, whereas the response time for users of Internet Explorer 5 showed a 50-millisecond improvement.

A 100-millisecond improvement (one-tenth of a second) seems small, but Amazon experimental data showed that a 100-millisecond increase in response time correlated to a 1% drop in sales. Understanding the direct link between revenue and performance can help justify the minimal investment required to switch from truecolor PNG to PNG8.

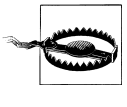
Based on this experiment, we recommend that you avoid `AlphaImageLoader` whenever possible. How can you avoid it? In the next section, we'll show some techniques to avoid using a filter altogether.

Progressively Enhanced PNG8 Alpha Transparency

If you've decided you just have to have alpha transparency, but you don't want the performance penalty associated with Microsoft's proprietary alpha filters, you can apply *progressive enhancement to PNG8*. This creates an image that uses alpha transparency where it's available, but doesn't rely on it. Follow these steps to achieve the best result:

1. Create a binary transparency image that uses only fully opaque or fully transparent pixels.
2. Write the CSS necessary to use the image.
3. Verify that the image works well without alpha transparency.
4. Add in the partially transparent pixels that will be displayed by better browsers. You can do this by layering the two image formats in Photoshop or your tool of choice, and saving the output as a separate file. To save the final image as PNG8 with alpha transparency you will need to use Fireworks or a command-line tool such as pngnq. We don't suggest allowing the software to autoconvert from truecolor PNG to PNG8, as the binary-transparent version is unlikely to be of acceptable quality.

To learn more about progressively enhanced PNG8, read Alex Walker's SitePoint article, "PNG8—The Clear Winner".



You can use PNG8 to progressively enhance images for which there is a clear binary transparent fallback. An image that has no fully opaque pixels would be rendered completely transparent by Internet Explorer 6.

For example, PNG8 progressive enhancement could be used on modules that overlay a variable background, such as the drop shadow popover above the Yahoo! Travel map shown in Figures 10-9 and 10-10.

Figure 10-9. Internet Explorer getting a simplified version with a clean 3-pixel border

Figure 10-10. Better browsers such as Internet Explorer 7 and 8, Firefox, Safari, and Opera also get a drop shadow

The human eye is very sensitive to variations in silhouette because humans identify objects, especially people, based on shape. Pay close attention to the edges of graphics. Mistakes are more noticeable when they affect the outline of an icon or image.

Optimizing Sprites

Dave Shea coined the term *CSS sprites* to refer to a process of combining multiple background images into one larger image and using background position to show or hide only a portion of that image in an HTML element (*http://www.alistapart.com/articles/sprites/*). The technique was later used by Yahoo! to improve performance by reducing the number of HTTP requests for the tiny icons on Yahoo!'s home page. There

are two approaches to optimizing sprites: the "everything and the kitchen sink" approach and the modular object-oriented approach. To figure out which is best for your site, ask the following questions:

- How many pages does your site have?
- Is your site modular? (Hint: it should be!)
- How much time can your team spend on site maintenance?

The answers can help you make the traditional trade-offs between the number of sprite(s), the maintenance cost, and the total number of unique pages. You can have any two, but not all three (see Figure 10-11).

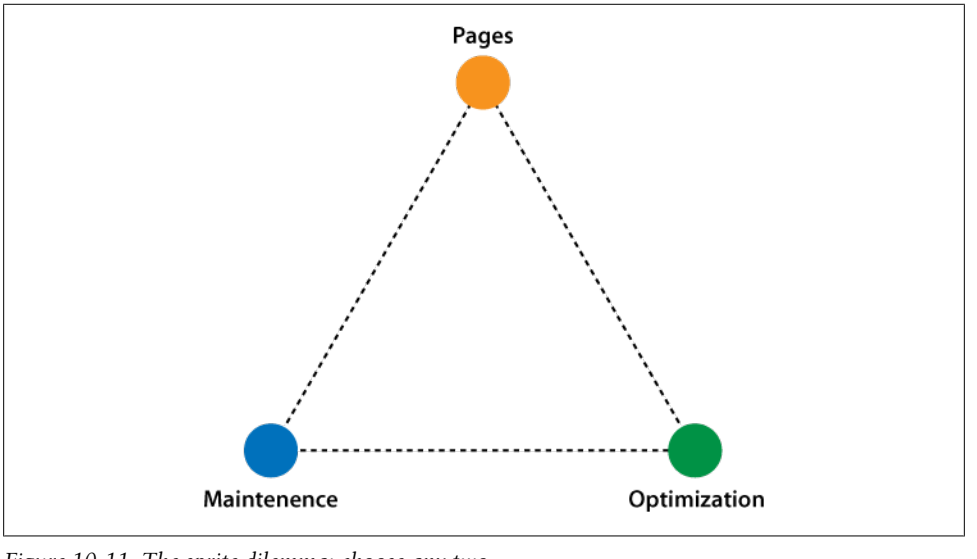


Figure 10-11. The sprite dilemma: choose any two

Über-Sprite Versus Modular Sprite

If a site has *very few pages*, it is best to sprite everything in the site into one über-sprite that includes all the images used on the site. Google uses an über-sprite for its search results page, as shown in Figure 10-12.

Figure 10-12. Google Search has only two pages; thus, it can have an über-sprite without significant maintenance costs

On the other hand, if your site has more pages, you need a different sprite strategy or the maintenance costs will be very expensive. The goal is to make it easy to remove stale modules from your site; otherwise, a once-performant site will become clunky. You can achieve this by spriting together images that belong to the same object. For example:

- All four corners from one rounded corner box
- The left and right sliding doors of your module headers
- The two to four images that make up a button
- Tab states such as current, hover, and normal

In a modular approach, these sprites would not be combined with other sprites.

Highly Optimized CSS Sprites

Sometimes optimizing sprites is more complex than optimizing images. Diverse resources combined in one sprite may be harder to compress well. Following these best practices will make your sprite as small as possible:

- *Combine like colors*; for example, sprite icons with a similar color palette.
- *Avoid unnecessary whitespace*, making images easier to process on mobile devices.
- *Arrange elements horizontally instead of vertically*. The sprite will be slightly smaller.
- *Limit colors* to stay within the 256-color limit of PNG8.
- *Optimize individual images, and then the sprite*. Color reduction will be easier with a limited palette.
- *Reduce anti-aliased pixels* via size and alignment. If an icon is slightly off-square, you can often reduce the anti-aliased pixels by aligning the image horizontally or vertically.
- *Avoid diagonal gradients*, which cannot be tiled.
- *Avoid alpha transparency in Internet Explorer 6 or quarantine images that require true alpha transparency in a separate sprite.*

- *Change the gradient color every two to three pixels*, rather than every pixel.
- *Be careful with logos.* They are very recognizable, so even small changes are likely to be noticed.

Other Image Optimizations

The remainder of this chapter discusses some additional image-related optimizations that can help your pages load faster. These concern how you use the image files rather than the images themselves.

Avoid Scaling Images

Unnecessary download overhead occurs when a 500 × 500-pixel image is scaled down in the HTML, like so:

```
<img src="image.jpg" width="100" height="100" alt="my image" />
```

This way, you cause the browser to scale down the image and show a smaller 100 × 100-pixel version of it. But the browser still needs to download the big image. You can achieve significant savings if you resize the image on the server side and serve the smaller version. As an additional selling point, be aware that some browsers don't do as good a job at scaling down as, for example, ImageMagick does, so the result of forcing the browser to do the scaling is degraded image quality *and* bigger downloads.

Crush Generated Images

If you're building a reporting application or module, chances are you'll need to generate different graphs or charts on the fly. When you generate those types of images, keep in mind the following two points:

- It is advisable to choose PNG over GIF, PNG8 being most preferred.
- Don't forget to crush the result before serving it.

An image found in the Google Chart API documentation makes a good example (see Figure 10-13). If you're not familiar with it, Chart is an excellent API that allows you to generate graphs by passing arguments in the URL. Let's take a look at how this service can be improved to generate images that are smaller.

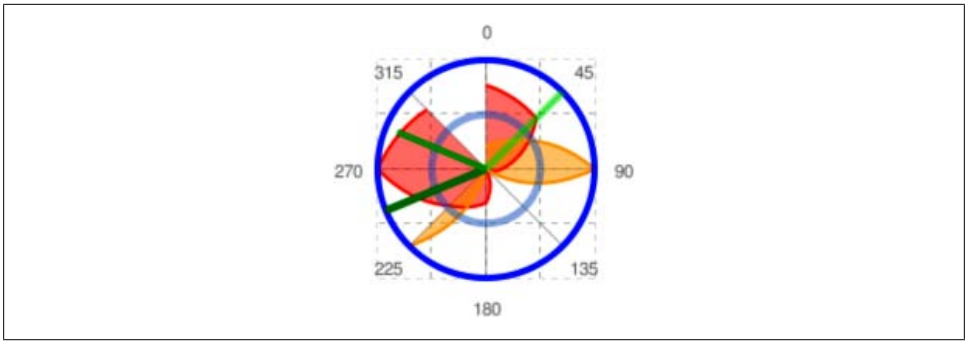


Figure 10-13. An example generated graph image

Figure 10-13 contains 1,704 colors, and the file size is 17,027 bytes. Two simple optimizations reduce the file size by more than half:

- Running this image through pngcrush results in an image that is 12,882 bytes, a savings of 24% with no loss in quality.
- Going one step further to convert the image to PNG8 using pngquant removes about 1,500 colors that the viewer doesn't notice. The new file size is 7,710 bytes, a savings of 55% from the original.

An additional benefit from writing the generated images to disk and crushing them is that when a second request for the same image arrives, you don't need to regenerate the image; you can serve the one that has already been cached and optimized.

Here is a simple piece of code that implements this advice using PHP with the GD image library:

```
<?php
header ('Content-type: image/png');

// name of the image file
$cachedir = 'myimagecache/';
$file = $cachedir . 'myimage.png';

// if in the cache, serve
if (file_exists($file)) {
    echo file_get_contents($file);
    die();
}

// new GD image
$im = @imagecreatetruecolor(200, 200);
// ... the rest of the image generation ...
imagepng($im, $file); // save
imagedestroy($im);    // cleanup

// crush the image
$cmd = array();
```

```
$cmd[] = "pngcrush -rem alla $file.png $file";
$cmd[] = "rm -f $file.png";
exec(implode(';', $cmd));

// spit out the new image
echo file_get_contents($file);
?>
```

Another option, instead of reading the file with `file_get_contents`, is to use a redirect to point the browser to the new location. Add an `Expires:` header for browsers that support redirect caching, because they'll reuse the image instead of downloading it on subsequent visits.

Favicons

Favicons are those small images named */favicon.ico* that sit in the web root and are displayed next to the URL in the browser's address bar (see Figure 10-14).

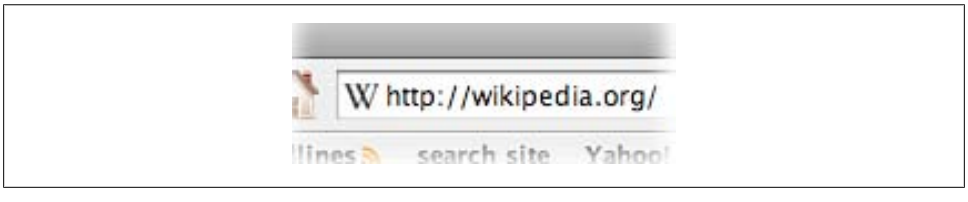


Figure 10-14. Wikipedia favicon

This page component is often ignored because it's small and supposedly cached. But caching is not as universal as we often think. This is true for any type of component, and favicons are no exception. Yahoo! Search noticed that it serves its favicon to 9% of all page views.

There are several points regarding favicons that significantly improve performance:

- Make sure to create a favicon. Since the browser will request this file anyway, there's no reason to return a 404 Page Not Found error, especially if your 404 handler consumes a database connection or other expensive resources.
- Consider adding an `Expires:` header when serving favicons. You cannot afford to cache "forever" if you serve the file from */favicon.ico* because you cannot rename the file if you decide to change it. But you can still cache for several months or even a year. Check the last modification date of your favicon file for an idea of how often you usually change it. And, if an emergency should arise, you can change the file name using a `<link>` tag, as explained next.
- You have an option to include the favicon using a `<link>` tag in the head. This way, you control the URL requested by the browser, as opposed to the predefined */favicon.ico*:

```
<link rel="shortcut icon" href="http://CDN/myicon.ico" />
```

This is great, because you can serve the favicon from a CDN and cache it "forever," sharing the same file among all your sites.

Be aware of one trade-off, though: if you do it this way, Firefox will request the favicon early in the waterfall, as opposed to at the very end after all other components are downloaded. On the other hand, if you serve the file from */favicon.ico*, there's no reason to add the `<link>` tag.

- Make the icon small. The ICO format can contain several images of different dimensions; for example, 16 × 16, 32 × 32, and so on. This will increase the file size of your icon, so it's best to use only one 16 × 16 image. This generally results in a file size of about 1 KB. As a rule of thumb, if your icon is bigger than 1 KB, you have room for improvement.
- Optimize the file with the free Windows utility called Pixelformer, experimenting with different palette sizes.

Apple Touch Icon

Similar to favicons are so-called Apple touch icon files used by iPhone/iPod devices. An Apple touch icon is just a PNG file in the root of your web server, 57 × 57 pixels in size, called *apple-touch-icon.png*. Again, if you want to serve this icon from a CDN and add a far-future `Expires:` header to it, you can use a `<link>` tag, like so:

```
<link rel="apple-touch-icon" href="http://CDN/any-name.png" />
```

Desktop browsers request this file much less often than they request favicons; the iPhone client will ask for it only when the user adds your page to his home screen.

Summary

In this chapter, you familiarized yourself with quite a few topics related to images, and you're now better prepared to ace your next image optimization project. Let's rehash some of the highlights:

- Start by choosing the appropriate format: JPEG for photos, GIF for animations, and PNG for everything else. Strive for PNG8 whenever possible.
- Crush PNGs, optimize GIF animations, and strip JPEG metadata from the images you own. Use progressive JPEG encoding for JPEGs more than 10 KB in file size.
- Avoid `AlphaImageLoader`.
- Use and optimize CSS sprites.
- Create modular sprites if your site has more than two to three pages.
- Don't scale images in HTML.
- Generated images should be crushed, too. Once generated, they should be cached for as long as possible. Convert images to PNG8 and determine whether 256 colors is acceptable.

- Don't forget favicons and Apple touch icons. Even if you don't refer to them in your HTML markup, they are still page components and should be small and cacheable.

CHAPTER 11

Sharding Dominant Domains

Some web pages have all their HTTP requests served from one domain. Other sites spread their resources across multiple domains. Rule 9 from *High Performance Web Sites* says to reduce DNS lookups, but sometimes increasing the number of domains is better for performance, even at the cost of adding more DNS lookups. The key is to find the web page's critical path. If the critical path results from too many resources being served from one domain, splitting them across multiple domains—what I call *domain sharding*—may make the page load more quickly.

Critical Path

Figure 11-1 shows the HTTP profile for eBay. The horizontal axis represents response time. A steep slope, as shown on the righthand side of the chart, reflects a lot of downloads in a short period of time. This is a sign of a fast page. In contrast, a flat slope such as the one shown in the first five HTTP requests means the browser is bogged down with a slow response or long-executing JavaScript. In this case, eBay's critical path is blocked by the HTML document in the first request, by JavaScript downloads in the fourth and fifth requests, and by JavaScript execution as indicated by the whitespace following the fourth and fifth requests.

Yahoo!'s HTTP profile, shown in Figure 11-2, has a different critical path. The majority of the time loading this page is spent downloading images two at a time.* All of the resources in the page are downloaded from a single domain: *l.yimg.com*. Some browsers, including Internet Explorer 6 and 7, limit the number of parallel downloads to two per server. (Internet Explorer 8 and Firefox 3 increase this to six per server, as discussed in "Newer Browsers" on page 169.) The impact of this two-per-server limit is evident in Figure 11-2—no more than two resources are downloaded in parallel at any given time. As a result, the HTTP profile forms a stair-step pattern that increases the time to load the page.

* This profile was produced using Internet Explorer 7.

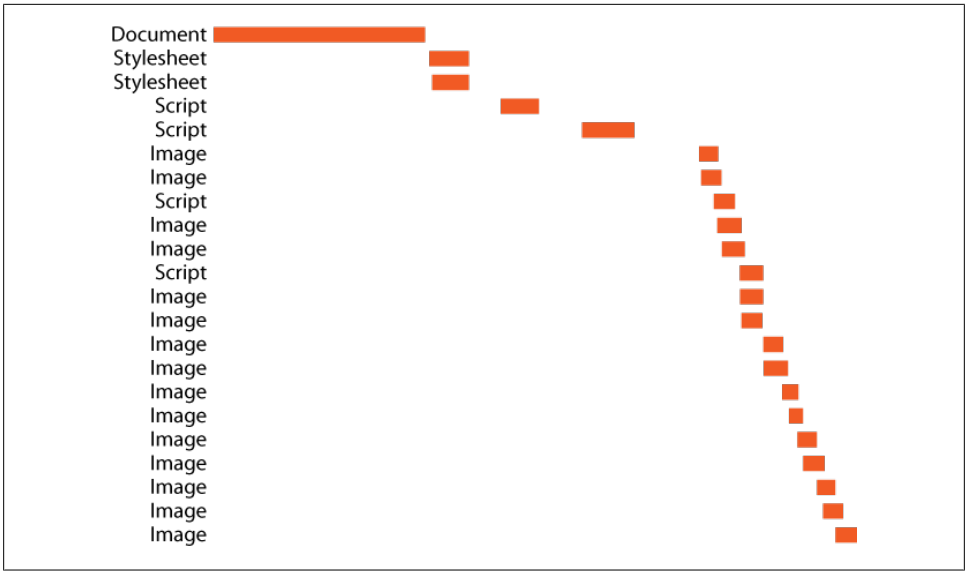


Figure 11-1. Critical path for http://www.ebay.com/

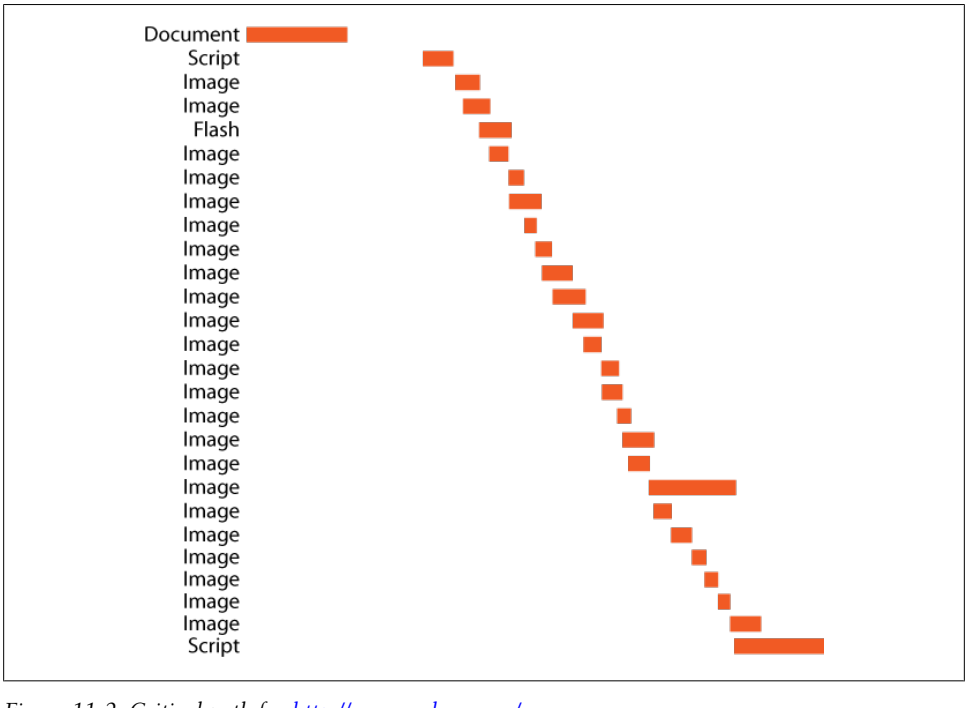


Figure 11-2. Critical path for http://www.yahoo.com/

When downloading resources from a single domain is the bottleneck, splitting resources across multiple domains speeds up the page by increasing the number of parallel downloads. This is demonstrated by the following examples.

One Domain
: *http://stevesouders.com/efws/domains1.php*

Two Domains
: *http://stevesouders.com/efws/domains2.php*

Each of these examples contains the 22 images from Yahoo!'s page. The One Domain example downloads all of the images from *l.yimg.com*, whereas the Two Domains example splits the images across two domains: *l.yimg.com* and *d.yimg.com*.† The Two Domains example loads 27% faster than the One Domain example (654 milliseconds versus 892 milliseconds over a 7,000 Kbps connection).

Figure 11-3 shows the HTTP profile for the One Domain and Two Domains examples. The waterfall at the top, in which only one domain is used, shows that only two resources are downloaded at any given time. In the waterfall at the bottom, on the other hand, we see four resources being downloaded simultaneously, which results in a faster-loading page.

Who's Sharding?

Table 11-1 shows which of the top web sites split resources across multiple domains. The total number of images, scripts, and stylesheets for each site is also shown.

Table 11-1. Use of multiple domains among top web sites

Web site	Images	Scripts	Stylesheets	Number of domains
http://www.aol.com/	59	6	2	3
http://www.ebay.com/	33	5	2	3
http://www.facebook.com/	96	14	14	10
http://www.google.com/search?q=flowers	3	1	0	N/A
http://search.live.com/results.aspx?q=flowers	6	1	4	5
http://www.msn.com/	45	7	3	3
http://www.myspace.com/	16	14	2	3
http://en.wikipedia.org/wiki/Flowers	33	6	9	2
http://www.yahoo.com/	28	4	1	1
http://www.youtube.com/	23	7	1	5

† I discovered that *d.yimg.com* was used by *http://news.yahoo.com* for downloading images.

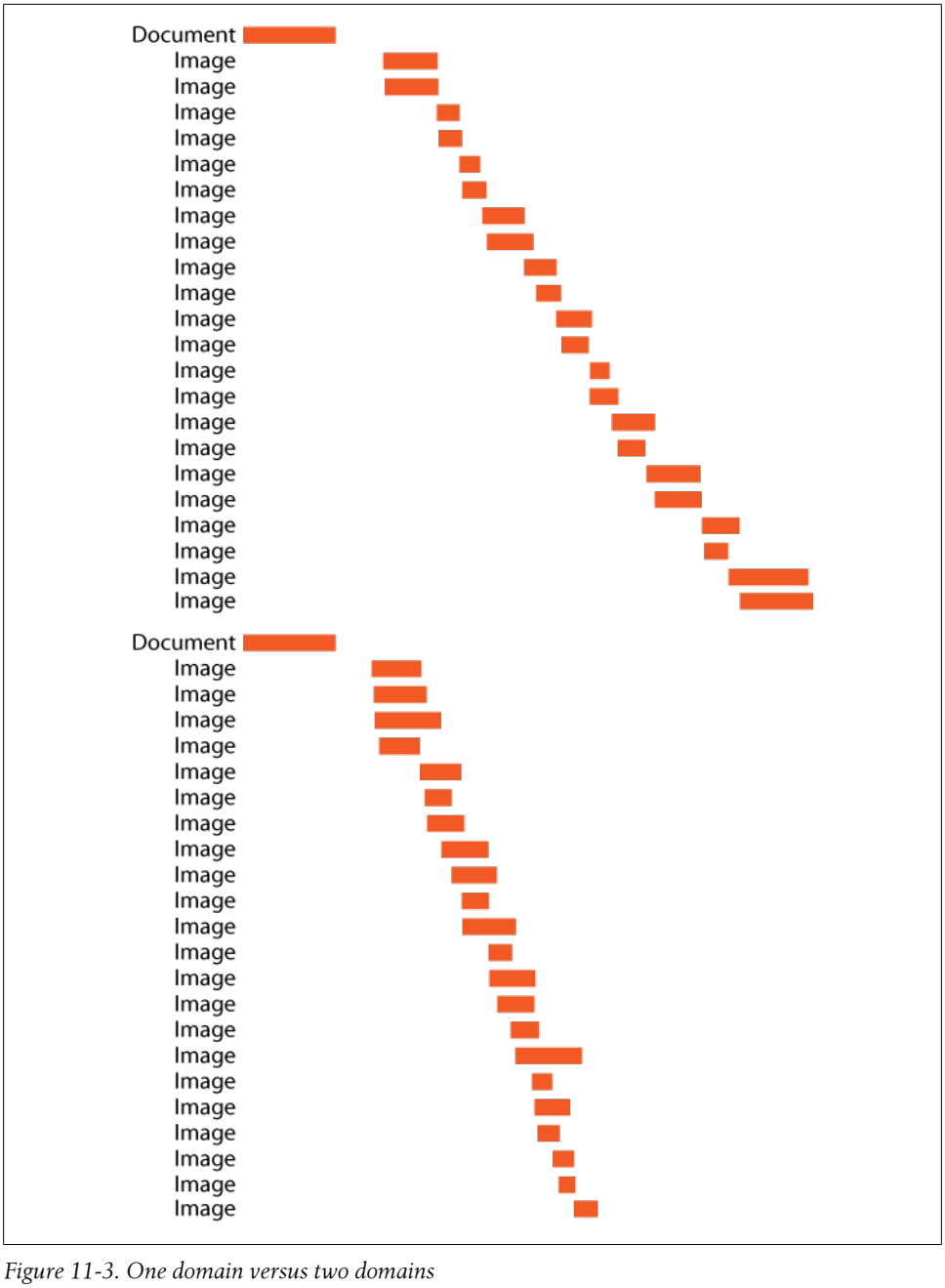


Figure 11-3. One domain versus two domains

Most of these sites shard their resources across multiple domains. It's especially clear that this is intentional for sites such as YouTube, where the domain names form a

sequence: *i1.ytimg.com*, *i2.ytimg.com*, *i3.ytimg.com*, and *i4.ytimg.com*. Many of these top sites have similar sequences in their sharded domains:

AOL
: *o.aolcdn.com*, *portal.aolcdn.com*, *www.aolcdn.com*

eBay
: *include.ebaystatic.com*, *pics.ebaystatic.com*, *rtm.ebaystatic.com*

Facebook
: *b.static.ak.fbcdn.net*, *external.ak.fbcdn.net*, *photos-[b,d,f,g,h].ak.fbcdn.net*, *platform.ak.fbcdn.net*, *profile.ak.facebook.com*, *static.ak.fbcdn.net*

Live Search
: *search.live.com*, *ts[1,2,3,4].images.live.com*

MSN.com
: *tk2.st[b,c,j].s-msn.com*

MySpace
: *cms.myspacecdn.com*, *rma.myspacecdn.com*, *x.myspacecdn.com*, *creative.myspace.com*, *largeassets.myspacecdn.com*, *x.myspace.com*

Wikipedia
: *en.wikipedia.org*, *upload.wikimedia.org*

YouTube
: *i[1,2,3,4].ytimg.com*, *s.ytimg.com*

Google's main page contains only two resources. They can be downloaded in parallel on one domain, so splitting across domains is not applicable. Yahoo! downloads most of its resources on one domain. It would benefit from splitting these across multiple domains. AOL and Wikipedia are an interesting story. They use just a few domains for a relatively large number of resources. One reason for this might be because they downgrade some of their responses from HTTP/1.1 to HTTP/1.0. The pros and cons of this are discussed in the following section.

Downgrading to HTTP/1.0

AOL and Wikipedia shard their resources across a relatively small number of domains. Even so, they achieve a high level of parallel downloads. Figure 11-4 shows the initial HTTP profile for Wikipedia when loaded in Internet Explorer 7. All of these resources are served from one domain: *en.wikipedia.org*. Internet Explorer 7 normally uses only two connections to a single server, but in this case we see that four connections are being used. This happens because Wikipedia downgrades its responses to HTTP/1.0.

HTTP/1.1 is used by most web clients and servers today, but HTTP/1.0 is still supported. When HTTP/1.1 is used, many browsers follow the limit of two connections per server as recommended in the HTTP/1.1 RFC. However, Internet Explorer 6 and 7 open more connections when HTTP/1.0 is used. The normal limit of two connections

per server is increased to four when HTTP/1.0 is used. Similarly, Firefox 2 uses two connections for HTTP/1.1, but increases that to eight connections in the presence of HTTP/1.0.

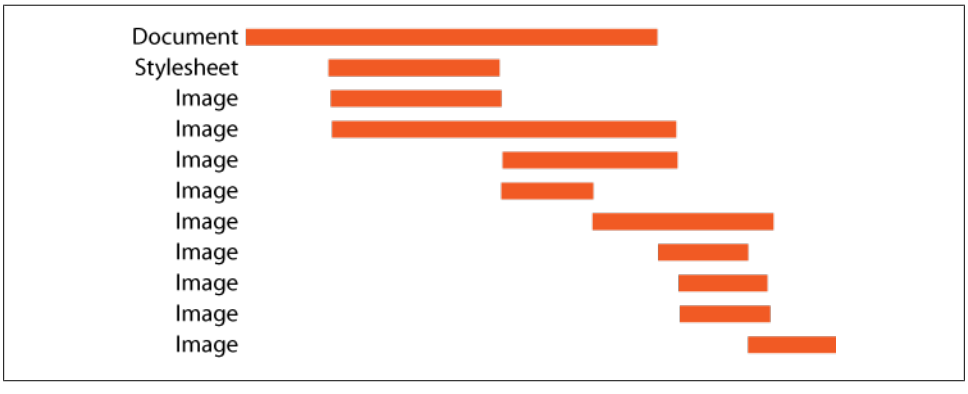


Figure 11-4. Wikipedia parallel downloads

A lower number of connections per server are recommended for HTTP/1.1 because of *persistent connections*. By default, HTTP/1.0 closes the TCP connection after each response. Establishing a new TCP connection for every request takes time. To reduce this overhead, HTTP/1.1 uses persistent connections and performs multiple requests and responses using a single connection. Persistent connections are typically held open longer and thus impose a greater burden on servers that have a finite number of connections available. Hence, the recommended number of connections per server is reduced to two for HTTP/1.1.

By downgrading to HTTP/1.0, AOL and Wikipedia achieve a higher level of parallel downloads, but this benefit is gained at the cost of losing persistent connections. Or is it? As an alternative to persistent connections, HTTP/1.0 supports the Keep-Alive option to reuse existing connections. There are differences between HTTP/1.0 Keep-Alive and HTTP/1.1 persistent connections, but they are subtle:

- Persistent connections are the default in HTTP/1.1. Once the HTTP version is specified as "HTTP/1.1," no additional header is necessary to declare support for persistent connections. But Keep-Alive is not the default for HTTP/1.0. Clients and servers must send the `Connection: Keep-Alive` header.
- There are risks involved in using HTTP/1.0 Keep-Alive connections through a proxy. A proxy that doesn't understand the `Connection: Keep-Alive` header and just blindly forwards it to the origin server might establish a hung connection while it waits for the origin server to close the connection. The origin server won't close the connection because it's establishing a Keep-Alive connection. Therefore, clients must be sure not to send `Connection: Keep-Alive` when talking to a proxy, which in fact all major browsers do.

- HTTP/1.0 Keep-Alive responses must use the `Content-Length` header to indicate the ending boundary between separate responses on a single connection. This means that dynamic content, where the total size is not known when the response is started, cannot take advantage of HTTP/1.0 Keep-Alive.
- Chunked transfer-encoding, introduced in HTTP/1.1, cannot be used with HTTP/1.0. Chunked encoding allows the server to send back data in chunks. This is most applicable for large responses generated dynamically, where the total size is not known but the server wants to start transferring the response as the content becomes available. (See Chapter 12 for more discussion about chunked encoding.)

These differences don't present any significant drawbacks to downgrading to HTTP/1.0 for static content. Popular browsers already send the `Connection: Keep-Alive` header and remove it when using a proxy. The size of *static* content is known when the request begins, so a `Content-Length` header can always be sent and there isn't a need for chunked encoding. It's possible that using chunked encoding for large resources, such as a 500 KB script, could result in faster downloading and parsing, but in practice none of the top sites use chunked encoding with their static content.

Users who access AOL and Wikipedia using Internet Explorer 6 and 7 benefit from the decision to downgrade to HTTP/1.0. They get resources downloaded four at a time and still benefit from reusing TCP connections thanks to Keep-Alive. Most other browsers, however, don't increase the connections per server based on HTTP version, as shown in Table 11-2.

Table 11-2. Connections per server

Browser	HTTP/1.1	HTTP/1.0
IE 6, 7	2	4
IE 8	6	6
Firefox 2	2	8
Firefox 3	6	6
Safari 3, 4	4	4
Chrome 1, 2	6	6
Opera 9, 10	4	4

If you have a large number of Internet Explorer 6 and 7 users, you might want to consider downgrading to HTTP/1.0. Doing so increases parallel downloads (for Internet Explorer 6 and 7) without the cost of an extra DNS lookup. But if you want all of your users to benefit from increased parallelization, domain sharding is the preferred solution.

Rolling Out Sharding

Several operational questions typically arise when considering splitting resources across multiple domains.

IP Address or Hostname

Browsers enforce the "maximum connections per server" constraint based on the hostname in the URL, not the IP address to which it resolves, as shown in the Different Hostnames, Same IP example.

Different Hostnames, Same IP
http://stevesouders.com/efws/hostnames.php

This example has four images: two from *stevesouders.com* and two from *www.stevesouders.com*. These hostnames have the same IP address. When loaded in Internet Explorer 6 and 7, all four images are downloaded in parallel. The browser has treated each hostname as a separate server, and consequently opened two connections for each one, even though these two hostnames resolve to the same IP address.

This is good news for people who want to split their content across multiple domains. It's not necessary to deploy additional servers. Instead, a CNAME record for the new domain can be used. A CNAME is just an alias from one domain name to another. Even though the domain names point to the same servers, the browser still opens the maximum number of connections for each unique hostname.

How Many Domains

In "Critical Path" on page 161, you saw that splitting content across two domains is better than splitting across one. Would three domains be better than two? How about 10? Research published by Yahoo! shows that increasing the number of domains from one to two improves performance, but increasing it above two has a negative effect on load times. The final answer depends on the number and size of resources, but sharding across two domains is a good rule of thumb.

How to Split Resources

Given a specific resource, what's the best algorithm for assigning it to one of multiple possible domains? A key feature of any splitting algorithm is that a specific resource always be assigned to the same domain. This ensures that, if the resource has already been cached, the URL for subsequent requests matches the URL in the cache.

One way to do this is to use a hashing function that converts the resource's filename into an integer that determines the chosen domain. Another alternative is to assign resources to domains based on the resource type. For example, stylesheets and images might be assigned to domain 1, while every other type of resource is assigned to domain 2. This might result in a lopsided distribution of resources across domains, but might actually be beneficial in that images could start downloading from domain 2 in parallel with stylesheets and scripts on domain 1.

Newer Browsers

Internet Explorer 8 and Firefox 3 both increase the number of connections per server from two to six. Striving to increase the number of parallel downloads for older browsers could result in too many parallel downloads for these next-generation browsers. If the browser opens too many connections, it could overload the server as well as degrade download efficiency on the client.

Sites that use several domains, such as Facebook and YouTube, may need to alter their splitting algorithm based on browser type. If you choose to split your static resources across multiple domains, follow the guideline of splitting across just two domains. This strikes a balance of improving performance for today's browsers as well as tomorrow's.

CHAPTER 12
Flushing the Document Early

The Performance Golden Rule reminds us to focus our performance improvements on the frontend—that's where most of the time is spent loading web pages.* Occasionally, there are exceptions to this rule where the backend takes a long time to generate the HTML document. Such a page might require intensive database queries or responses from other web services before the HTML content is returned.

Unfortunately, while the backend chugs away, everything on the user's end is on hold. Rather than letting the browser sit idle and leaving the user waiting for feedback, this chapter explains how to start the page loading even before the HTML document is completed.

Flush the Head

In most cases, the browser waits for the HTML document to arrive before it starts rendering the page and downloading the page's resources. This is shown by the Simple Page example.

Simple Page
http://stevesouders.com/efws/simple.php

This example page contains two images and a script. The HTML document and its three resources are all programmed to take two seconds to return. The HTTP waterfall chart for Simple Page is shown in Figure 12-1. As expected, the HTML document is downloaded first. Once it arrives, the browser parses the HTML, renders the first few lines of text, and starts downloading the page's resources.

* See Rule 1 from *High Performance Web Sites*.

Figure 12-1. Simple Page HTTP waterfall chart

The two images, served from *1.cuzillion.com*, are downloaded in parallel. The script is also downloaded in parallel, because it comes after the images and is hosted on a different hostname, *2.cuzillion.com*. The overall page load time is four seconds.

Figure 12-2 shows the same page, but in this case the images and script start downloading before the HTML document has fully arrived. The result is a page that takes only two seconds to load—half the time of the original example.

Figure 12-2. Flush HTTP waterfall chart

This speedup was achieved by adding a call to PHP's `flush` function. Running the Flush example makes it clear that this is a much faster page.

Flush
http://stevesouders.com/efws/flush-nogzip.php

To understand how `flush` works, and the complications that ensue, we need to understand how HTML documents are generated.† As the server parses the PHP page, all output is written to STDOUT. Rather than being sent immediately, one character, word, or line at a time, the output is queued up and sent to the browser in larger chunks. This is more efficient because it results in fewer packets being sent from the server to the browser. Each packet sent incurs some network latency, so it's usually better to send a small number of large packets, rather than a large number of small packets.

Calling `flush()` causes anything queued up in STDOUT to be sent immediately. However, simply flushing STDOUT isn't enough to achieve the type of speedup experienced in the preceding example. The call to `flush` has to be made in the right place. Let's look at the PHP source code for the Flush example:

† Although this discussion focuses on PHP, these concepts are applicable to other templating frameworks as well.

```
<html>
<body>

<p>
This is the Flush example.
</p>

<img src="http://1.cuzillion.com/...">
<img src="http://1.cuzillion.com/...">
<script src="http://2.cuzillion.com/..." type="text/javascript"></script>

<?php
flush();
long_slow_function();
?>

<p>
This sentence is after the long, slow function.
</p>

</body>
</html>
```

Remember, the motivation for this chapter is those occasions when you have an HTML document that takes a long time to generate. That is represented in our PHP code by the call to `long_slow_function` (in this case a two-second sleep). The call to `flush()` is made right *before* this long backend delay.

Whether this speeds up the page depends on what you include in the HTML before the call to `flush()`. In this example, there is a line of text ("This is the Flush example") and three resources (two images and one script). This is exactly what's needed to combat the two shortcomings of a slow HTML document: blocked rendering and blocked downloads. By including the line of text in the flushed HTML, the user is given visual feedback that the page is loading. By including the three resources in the flushed output, the browser starts downloading resources even while it waits for the rest of the HTML document. This is the key performance insight from this chapter; getting resources downloading early is the primary benefit that flushing provides.

This seems pretty simple, yet reading the comments on the `flush` documentation page reveals that it's not as simple as it looks.

Output Buffering

Perhaps the biggest confusion around getting flushing to work in PHP involves output buffering. As explained earlier, PHP output is written to STDOUT. *Output buffering* adds another layer where output is queued before it goes to STDOUT.

The first step is to determine whether output buffering is turned on in your PHP configuration, and how big the buffer is. This is controlled by the `output_buffering` directive in *php.ini*.[‡] Part of the confusion stems from the fact that the default value changed in PHP 4.3.5. Before that, output buffering was enabled by default with a buffer size of 4,096 bytes, equivalent to this line in your *php.ini*:

```
output_buffering = 4096
```

With PHP 4.3.5, the default changed to output buffering being disabled:

```
output_buffering = 0
```

You can use this PHP code to find out the value of `output_buffering` and what version of PHP is running on your server:

```
<?php
echo "<br>output_buffering = " . ini_get('output_buffering');
echo "<br>PHP version = " . phpversion();
?>
```

If output buffering is on for your server, in addition to using `flush`, you'll also have to use `ob_flush` and its related functions, as shown by the Flush Output Buffering example.

Flush Output Buffering
: *http://stevesouders.com/efws/ob/flush-nogzip-ob.php*

The PHP code, with the added lines in bold, is shown in the sample that follows. The most intuitive new function calls are `ob_start` and `ob_flush`; `ob_start` opens a new output buffer while `ob_flush` flushes the contents of this output buffer to STDOUT. Once the output buffer is flushed, we still need the call to `flush()` in order to flush STDOUT:

```
<?php
while (ob_get_level() > 0) {
    ob_end_flush();
}
ob_start();
?>
<html>
<body>

<p>
This is the Flush Output Buffering example.
</p>

<img src="http://1.cuzillion.com/...">
<img src="http://1.cuzillion.com/...">
<script src="http://2.cuzillion.com/..." type="text/javascript"></script>

<?php
ob_flush();
flush();
```

[‡] *http://www.php.net/manual/en/outcontrol.configuration.php#ini.output-buffering*

```
long_slow_function();
?>

<p>
This sentence is after the long, slow function.
</p>

</body>
</html>
```

The `while` loop calls `ob_end_flush` as long as `ob_get_level()` is greater than zero. Forgetting this step is where many developers go wrong. This loop ensures that any output buffers that are already open are flushed and removed. Without doing this, the call to `ob_start` might not be the only output buffer opened. Output buffers in PHP are stacked. If our call to `ob_start` opened a second output buffer, the subsequent call to `ob_flush` would flush this second output buffer into the first output buffer, but not to STDOUT. Consequently, the call to `flush` would have no effect, since STDOUT would be empty. Confirming whether output buffering is enabled and, if so, using the `ob_` functions solves these issues.

Chunked Encoding

The flush examples won't be faster when using web servers or clients that support only HTTP/1.0. That's because they don't support chunked encoding.

HTTP/1.0 responses are returned as one block of data, the size of which is communicated in the `Content-Length` header. The browser needs to know the size of the data in order to know when the response ends. Because the HTML document is sent as one block, the browser can't start rendering the page and downloading resources until the whole response arrives.§

HTTP/1.1 introduced the `Transfer-Encoding: chunked` response header.‖ With *chunked encoding*, the HTML document can be returned in multiple blocks of data. Each chunk of the response starts with its own size indicator. This allows the browser to parse each chunk as soon as it arrives, resulting in a page that loads faster.

Chunked encoding fosters faster pages in two other ways, both related to dynamic pages. Without chunked encoding, responses must contain a `Content-Length` header. This means the server can't start sending the response until it finishes stitching the entire response together and measuring its size. With chunked encoding, the server can start transmitting the response sooner, because it only needs to know the size of each chunk being sent.

§ A possible alternative if `Content-Length` is not known is for the server to close the connection, but this defeats the benefits of persistent connections.

‖ *http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.41*

The second performance opportunity made possible by chunked encoding comes by use of the `Trailer` header.[#] In some situations, it's not possible to know whether a header is needed or what its value should be until the HTML document has been created. For example, during generation of the HTML content, results from a database query or web service request might determine the value for a `Cookie` or `ETag` response header.

Normally, these headers must be sent in the beginning of the response, which means the server can't start sending the response until these time-consuming database queries or web service calls have completed. Alternatively, when chunked encoding is used, these headers can be sent later. The first chunk is sent immediately and uses the `Trailer` header to list the headers that will come later:

```
Trailer: Cookie
Trailer: ETag
```

The `Cookie` and `ETag` headers can then be included at the end of the HTML response.[*]

Chunked encoding makes it possible to start sending parts of the HTML document immediately, even before the total size and other headers are known. To gain the benefits of flushing the document early, you'll need to make sure chunked encoding is working. Fortunately, Apache and other web servers take care of this for you. If you're trying to get flushing to work, make sure to confirm the presence of `Transfer-Encoding: chunked` in the HTML document's response headers.

Flushing and Gzip

In the previous example, the HTML document was not gzipped. Gzipping the HTML document is critical for all web sites, but it adds another level of complexity in getting flushing to work. If the earlier flush examples are gzipped, the flushing doesn't work, as the Flush Gzip No Padding example shows.

Flush Gzip No Padding
: *http://stevesouders.com/efws/flush-gzip-no-padding.php*

In this case, flushing is thwarted because of how Apache buffers output when compression is enabled. Apache 2.*x* uses `mod_deflate` for compression.[†] This module has a buffer that is 8,096 bytes by default. You can reduce the size of this buffer using the `DeflateBufferSize` directive. If that's not an option, you can make flushing work if you add more than 8 KB of padding (after compression) to the HTML document, as shown by the Flush Gzip Padding example.

[#] *http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.40*

[*] As of this writing, browser support for trailers is mixed. More research and evangelism are needed to ensure cross-browser support.

[†] *http://httpd.apache.org/docs/2.0/mod/mod_deflate.html*

Flush Gzip Padding
: *http://stevesouders.com/efws/flush-gzip-padding.php*

Adding the padding causes the deflate buffer to fill up and get flushed to the browser. The padding is tricky. Normally, you could use PHP's `str_pad` function:

```
echo str_pad('', 20000);
```

That won't work in this situation; compression is enabled, so the padding is compressed to less than the necessary 8 KB. The Flush Gzip Padding example has a 20 KB comment of nonrepeating strings, so even when compressed it exceeds the 8,096-byte size of the deflate buffer, allowing the flush to proceed.

Adding 20 KB to your pages is a high price to pay. Luckily, Apache 2.2.8 and later fix this issue and don't require this padding trick. At the time of this writing, the company hosting my web site is still running Apache 2.0. I tested this on servers using Apache 2.2.8 and confirmed that the page is compressed and flushing works, even without the padding.

Other Intermediaries

Proxies and antivirus software are two intermediaries that have the potential to obstruct the performance benefits of flushing. If these intermediaries are used to filter content, rather than forwarding the flushed blocks of data, they may wait until the entire response is received and scanned before forwarding on to the web client.

Another issue involves proxies that downgrade all responses to HTTP/1.0; since HTTP/1.0 doesn't support chunked encoding, flushing doesn't work with these proxies. One example is the Squid proxy. Its Wiki states that HTTP/1.1 is not yet supported, and one of the major reasons is "Chunked encoding [is] missing".‡

I've seen developers spend hours debugging why flushing wasn't working, only to realize they were connected to the Internet through a company proxy that breaks flushing. To determine whether you're behind a proxy, you can check your browser's network connection settings. Sometimes it's hard to tell whether you're set up to use a proxy, especially with configuration options such as "Automatically detect settings" (Internet Explorer) and "Auto-detect proxy settings for this network" (Firefox 3.0). In addition to checking my browser settings, I look for headers such as `Proxy-Connection`, `X-Forwarded-For`, and `Via` or a status containing "HTTP/1.0" in the HTML document response. If any of these are present, you're probably going through a proxy that may prevent flushing from working.

‡ *http://wiki.squid-cache.org/Features/HTTP11*

Domain Blocking During Flushing

Mainstream browsers, such as Internet Explorer 6 and 7 and Firefox 2, support only two connections per server. This limits the number of parallel downloads that can be performed on any single domain. There are ways to work around this limitation, as described in Chapter 11. Most of the issues with domain blocking arise in the context of resources in the page. With flushing, however, we also need to keep in mind that the HTML document request can affect parallel downloads.

In the examples so far, I've been careful to put the two images on a different domain (*1.cuzillion.com*) than the main page (*stevesouders.com*). Why is this important? There are only two images, so even in Internet Explorer 7 (with only two connections per server) there shouldn't be any blocking behavior.

This sounds right, but as soon as we see the results we understand that because we're using chunked encoding, the HTML document response is still using one connection—this leaves only one other connection for other resources that are on the same domain. The Flush Domain Blocking example creates this situation—in it the two images are served from *stevesouders.com*.

Flush Domain Blocking
http://stevesouders.com/efws/flush-domain-blocking-nogzip.php

Loading this in Internet Explorer 7 results in the HTTP waterfall chart shown in Figure 12-3. Comparing this to Figure 12-2, we see that this page is almost twice as slow, even though flushing is working. The reason it's slow is that the second image is blocked until the HTML document returns. The HTML document and the first image have used up the two connections to *stevesouders.com*, so the second image has to wait.

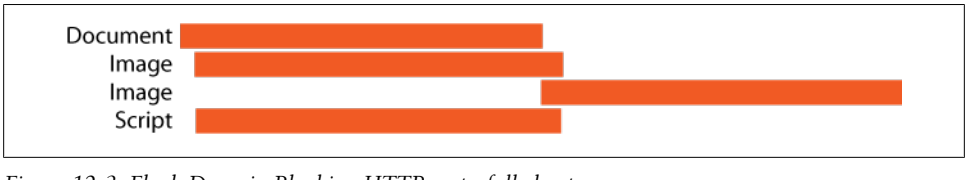


Figure 12-3. Flush Domain Blocking HTTP waterfall chart

If you leverage flushing's greatest benefit of downloading resources early, it's easy to exceed the two connections allowed to a single server. Keep this in mind as you might need to fetch the initial resources from a domain that's not blocked by the HTML document.

Browsers: The Last Hurdle

As if getting flushing to work wasn't hard enough, you'll get thrown off track if you're using Safari or Chrome with these examples. Even though both support chunked

encoding, they won't start rendering the page until they've received a minimum threshold of data: Safari is around 1 KB, Chrome is around 2 KB.[§]

The Flush example has only ~600 bytes in the first chunk, and as a result, the benefits of flushing aren't seen in Safari and Chrome. To get flushing to work in Safari, the Flush 1K example includes an additional 1 KB of HTML in the initial chunk. Similarly, the Flush 2K example includes an extra 2 KB of HTML and works in Chrome.

Flush 1K
: *http://stevesouders.com/efws/flush-nogzip-1k.php*

Flush 2K
: *http://stevesouders.com/efws/flush-nogzip-2k.php*

This is an issue in my examples because the HTML is minimal. In real-world pages, with inline style and script blocks and more page markup, it's likely that the flushed HTML exceeds 2 KB. If you want flushing to work across all browsers, make sure you flush after the 2 KB mark.

Flushing Beyond PHP

This chapter, as well as most wikis and forums related to flushing, focuses on PHP. If you use a different HTML templating framework, don't fear; there's probably a way to get the performance gains from chunked encoding, and it's probably done with a function named "flush."

Experienced Perl programmers at some point have written scripts where the printed output needed to be flushed from STDOUT immediately. These aficionados know that setting `$|` to a nonzero value is the way to accomplish this. Lesser known is the technique for doing this with the `FileHandle autoflush` method.[‖] The Flush Perl example uses this technique.

Flush Perl
: *http://stevesouders.com/efws/flush-nogzip.cgi*

The call to `autoflush` is at the top of the script:

```
use FileHandle;
STDOUT->autoflush(1);
```

Python file objects[#] and Ruby's `IO` class[*] have a `flush` function. It's likely that no matter what language you're using on the backend, there's a way to flush STDOUT.

[§] Internet Explorer has a similar minimum threshold, but it's only 255 bytes, so it's unlikely to trip you up.

[‖] *http://perldoc.perl.org/FileHandle.html*

[#] *http://www.python.org/doc/2.5.2/lib/bltin-file-objects.html*

[*] *http://www.ruby-doc.org/core/classes/IO.html#M002303*

The Flush Checklist

Getting flushing to work isn't always easy. If you're trying to flush your PHP page, but are having trouble getting it to work, here's the checklist to walk through:

- Is output buffering on? If so, you have to use the `ob_` functions.
- Do you see the `Transfer-Encoding: chunked` response header? Chunked encoding is typically required for flushing to work.
- Is the response gzipped? If so, and you're running a version of Apache earlier than 2.2.8, you have to add padding to your page.
- Are you behind a proxy or using antivirus software? These might buffer the chunks before sending them through to the browser.
- Are any of the resources referenced in the flushed chunk being blocked because they're fetched from the same domain as the HTML document?
- Are you testing only in Safari or Chrome? The flushed HTML must be more than 2 KB to see the benefits in these browsers.

There are many variables to sort out—it's almost like trying to get the stars to align—but the results are worth it. In our set of top 10 web sites, 5 use chunked encoding: AOL, Facebook, Google Search, MySpace, and Yahoo!. Keep in mind that even though these sites support chunked encoding, it's not necessarily the case that they are flushing the document early. The HTTP profile for Google Search, in Figure 12-4, most clearly shows the benefits of flushing.

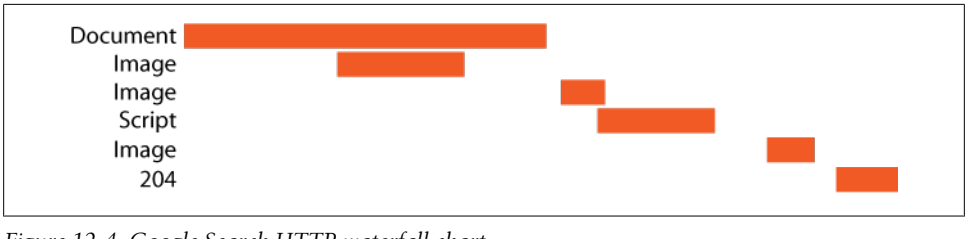


Figure 12-4. Google Search HTTP waterfall chart

By flushing the document early, Google pages start downloading resources and rendering content more quickly. This is a benefit that all users will appreciate, especially those with slow Internet connections and high latency.

CHAPTER 13
Using Iframes Sparingly

Iframes, also called inline frames, allow for one HTML document to be embedded inside another.* Iframes are best used for integrating HTML content, such as an ad, that's from a web site different from that serving the main page.

A benefit of using iframes for this purpose is that their document is entirely independent from the parent document. Relative URLs inside the iframe are resolved relative to the iframe's base URI, not the parent's. User agents can give the iframe focus for printing, bookmarking, saving, and so forth. Perhaps most important, JavaScript included in the iframe has limited access to the parent. For example, an iframe from a different domain can't access the parent's cookies. This is an important consideration when web developers must allow third-party content, such as ads, in their pages but they don't have control over this content.

What's the downside? You guessed it—slower performance. Chapter 4 describes how iframes are used to improve performance in terms of loading scripts asynchronously. It is true that iframes can make pages load faster, if used properly. Unfortunately, iframes are often used in a way that hurts performance. It's important to know the performance penalties inflicted by iframes and how to avoid them.

The Most Expensive DOM Element

The Cost of Elements test page measures how long it takes to create different types of DOM elements.

Cost of Elements
http://stevesouders.com/efws/costofelements.php

Using this page, I tested how long it takes to load 100 elements of the following types: `A`, `DIV`, `SCRIPT`, `STYLE`, and `IFRAME`. I ran each test 10 times in Chrome (1.0, 2.0), Firefox (2.0, 3.0, 3.1beta2), Internet Explorer (6, 7, 8beta2), Opera (9.63, 10.00alpha),

* *http://www.w3.org/TR/html4/present/frames.html#edef-IFRAME*

and Safari (3.2, 4.0 developer preview). Figure 13-1 shows the median time to create 100 elements of each type.

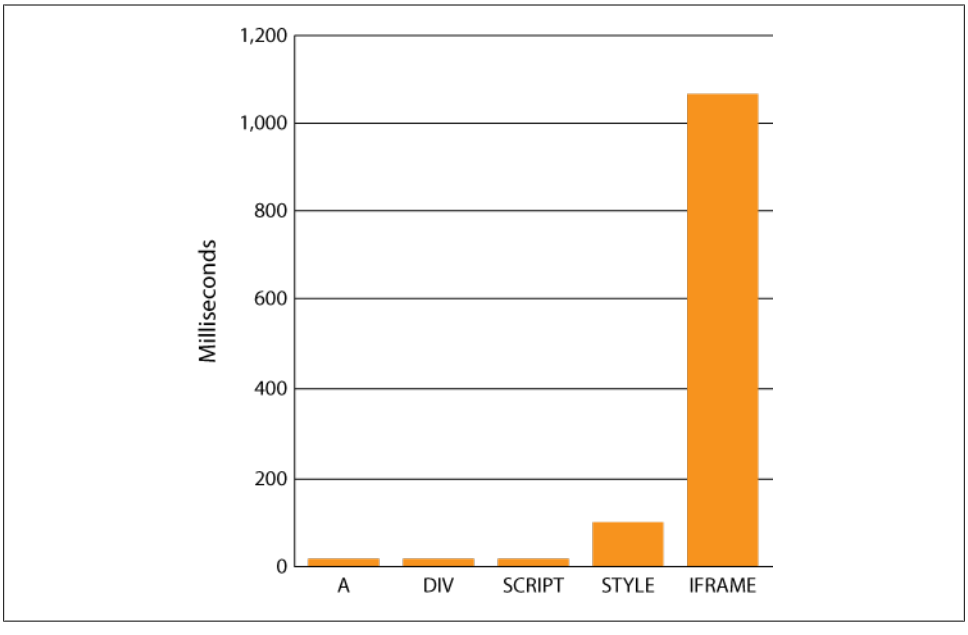


Figure 13-1. Time to load 100 elements of various types

Figure 13-1 shows that iframes are one to two orders of magnitude more expensive to create than other types of DOM elements. In these tests, the DOM elements are empty. It's possible that a large script or style block could take longer to load than certain types of iframes, but this test compares the baseline cost. Given the much larger cost of iframes, they should be used in small numbers with caution.

Iframes Block Onload

We want the `onload` event to fire as quickly as possible. There are several reasons for this:

- When `onload` fires, the browser's busy indicators stop and the user is given feedback that the page is ready. For example, "Done" is displayed in the status bar. Thus, if `onload` fires quickly, there's a greater chance that the user perceives the page to be fast.
- Developers frequently initiate UI actions with the `onload` event; for example, setting focus to a login field. As users become trained to wait for this action, it's important for `onload` to fire as soon as possible so that the user's wait is short.

- Developers sometimes associate important actions with the window's unload event; for example, JavaScript code to reduce memory leaks.† Unfortunately, in some browsers, `onunload` will not fire unless the `onload` event has fired.‡ If `onload` takes too long and the user quickly leaves the page, then the `onunload` code never executes.§

It makes sense that the `onload` event shouldn't fire until all critical content on the page has loaded, but often an iframe contains content that is not critical to the user's engagement with the page. A good example of this is iframes that contain ads. Ads may be critical to the web site's business, but the user experience should not be degraded waiting for ads to load. When used in the typical fashion, iframes block the `onload` event. It's important, therefore, to investigate whether there's a way to load iframes without delaying the main page's `onload` event.

The Iframe Blocking Onload example shows that iframes block the parent window's `onload` event.

Iframe Blocking Onload
: *http://stevesouders.com/efws/iframe-onload-blocking.php*

In this example, the iframe is used in the typical way, setting the iframe's URL with the HTML `SRC` attribute, like so:

```
<iframe src="url"></iframe>
```

There are four variations of this example, accessible through the links in the page:

Empty iframe
: The iframe takes four seconds to return but doesn't contain any resources.

Iframe with image
: The iframe returns immediately but contains an image that takes four seconds to return.

Iframe with script
: The iframe returns immediately but contains an external script that takes four seconds to return.

Iframe with stylesheet
: The iframe returns immediately but contains a stylesheet that takes four seconds to return.

The parent window's onload time is shown at the top of the page. Since the parent window contains only one resource (the iframe), we know the iframe has blocked the

† "Understanding and Solving Internet Explorer Leak Patterns," *http://msdn.microsoft.com/en-us/library/bb250448.aspx*.

‡ This is true in Internet Explorer 6 through 8, Safari 3 and 4, and Chrome 1 and 2.

§ There are workarounds to this problem—for example, *http://blog.moxiecode.com/2008/04/08/unload-event-never-fires-in-ie/*—but there will continue to be developers, unaware of the issue, who use the unload event.

`onload` event if the onload time is greater than four seconds. The result is the same across all major browsers: *iframes block the parent window's* `onload` *event when used in the typical way.*

There's an easy workaround to this blocking behavior, but it works only in Safari and Chrome. Instead of setting the iframe's URL with the HTML `SRC` attribute, we set it dynamically with JavaScript:

```
<iframe id=iframe1 src=""></iframe>
<script type="text/javascript">
document.getElementById('iframe1').src = "url";
</script>
```

This technique is used in the Iframe Not Blocking Onload example.

Iframe Not Blocking Onload
http://stevesouders.com/efws/iframe-onload-nonblocking.php

In Safari and Chrome, the onload time is a few hundred milliseconds. Since this is much less than four seconds, we know that the iframe and its components are not blocking the parent window's `onload` event. Unfortunately, this technique doesn't work in Internet Explorer, Firefox, and Opera. For a majority of users, the blocking behavior of iframes prolongs the time before the page is "Done."

Parallel Downloads with Iframes

This section explores the download behavior of iframes and the main page. In general, resources in an iframe are downloaded in parallel with resources in the main page. In some cases, however, the main page can cause resources in the iframe to be blocked from downloading.

Script Before Iframe

External scripts in the main page that are loaded in the typical way (`<script src="url"></script>`) block all resources that follow. Therefore, an iframe and its resources are blocked from downloading if they are preceded by an external script. This is demonstrated in the Script Before Iframe example.

Script Before Iframe
http://stevesouders.com/efws/script-before-iframe.php

In this example, the script takes four seconds to return. The iframe has no delay, but it contains an image, a stylesheet, and a script that are each configured to take four seconds to return. Figure 13-2 shows the HTTP waterfall charts for this example in Internet Explorer, Firefox, Safari, Chrome, and Opera. (Safari, Chrome, and Opera perform similarly and so are grouped together.) As expected, we see that the script in the main page blocks the iframe request; this causes the iframe's resources to be delayed. The way scripts block iframes is similar across all browsers, but in the next two

sections we'll see that Internet Explorer and Firefox diverge from the behavior shown here, while Safari, Chrome, and Opera all perform the same.

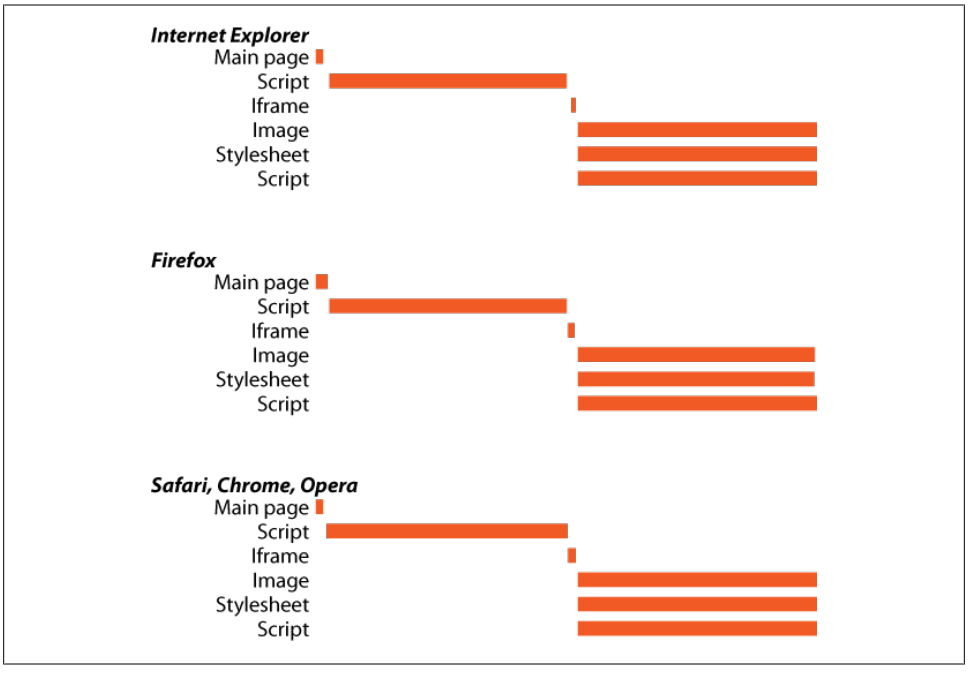


Figure 13-2. Script before iframe

Stylesheet Before Iframe

Chapter 7 discusses the blocking behavior inflicted on pages that contain a stylesheet followed by an inline script. Stylesheets also have an unexpected blocking interaction with iframes in Internet Explorer and Firefox. The Stylesheet Before Iframe example shows what happens.

Stylesheet Before Iframe
http://stevesouders.com/efws/stylesheet-before-iframe.php

Normally, stylesheets don't block other resources, and in Figure 13-3 we see that this is true for Safari, Chrome, and Opera. However, in Internet Explorer and Firefox, the stylesheet blocks requests associated with the iframe. In Internet Explorer, the iframe request is blocked. In Firefox, the stylesheet and iframe download in parallel, but the iframe's resources are blocked by the stylesheet.‖

‖ Performance in Firefox 2 is worse because stylesheets block all downloads. This was fixed in Firefox 3.0.

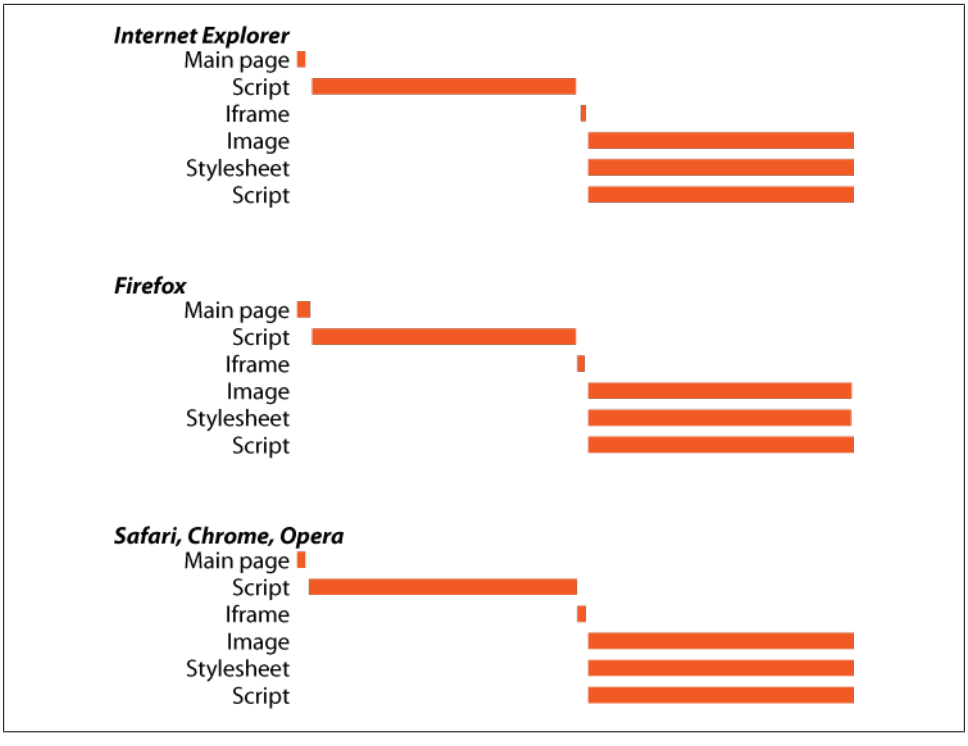


Figure 13-3. Stylesheet before iframe

Stylesheet After Iframe

Moving the stylesheet below the iframe would presumably avoid this blocking behavior. This is true in Internet Explorer, but not in Firefox, as shown by the Stylesheet After Iframe example.

Stylesheet After Iframe
http://stevesouders.com/efws/stylesheet-after-iframe.php

Firefox's waterfall chart, as shown in Figure 13-4, takes eight seconds to load. The other major browsers are all down to four seconds. Although it's not worthwhile to move stylesheets lower in the page—any gains from not blocking iframes are lost because rendering is delayed—it is worth noting that if the iframe's resources were in the main page itself, the blocking wouldn't happen. When deciding whether iframes are an appropriate solution, this blocking behavior is an important consideration.

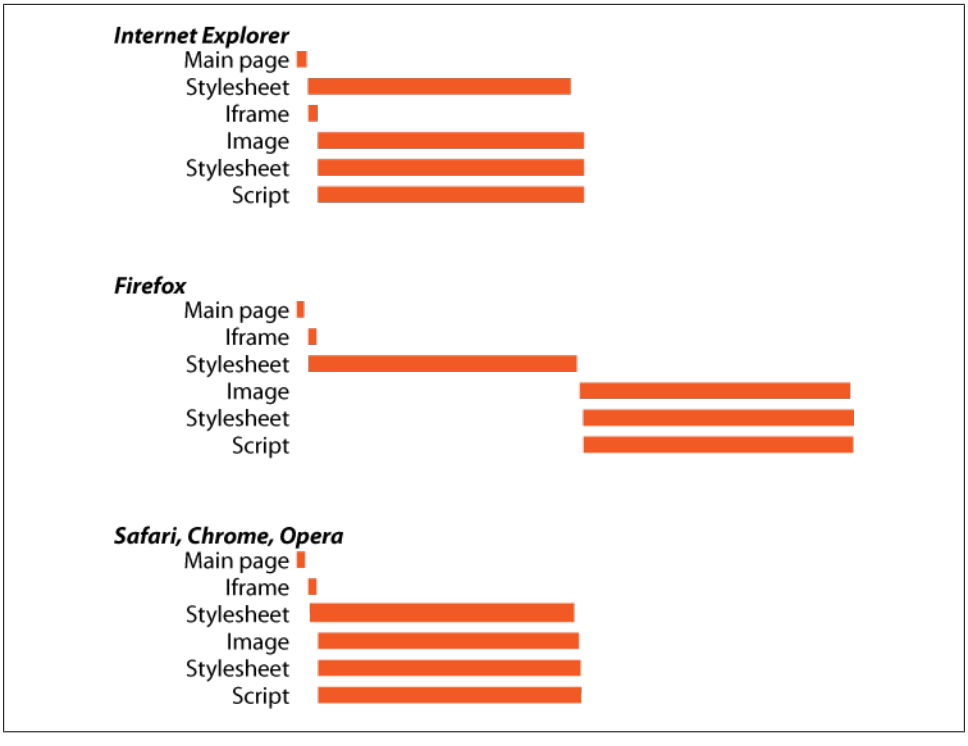


Figure 13-4. Stylesheet after iframe

Connections per Hostname

Browsers have a limit on the number of connections they open to a single hostname. The number of connections determines how many resources can be downloaded in parallel. Internet Explorer 6 and 7 and Firefox 2 open only two connections per server. Newer browsers open a higher number—between four and eight connections per server. (See Table 11-2 for a full breakout by browser.) The following sections explore how these browsers enforce these limits across iframes, tabs, and windows.

Connection Sharing in Iframes

One might hope that because an iframe is "entirely independent of the document in which it is embedded,"[#] resources downloaded as part of the iframe would use a connection pool that is separate from the main page. The Iframe Connections example tests whether this is true.

[#] *http://www.w3.org/TR/html4/struct/objects.html#h-13.5*

Iframe Connections

http://stevesouders.com/efws/parent-connections.php

The Iframe Connections example downloads five images in the parent document and another five images as part of the iframe. All 10 images are from the same server (*1.cuzillion.com*) and are configured to take two seconds to respond. Figure 13-5 shows the HTTP waterfall chart for this example loaded in Internet Explorer 7.

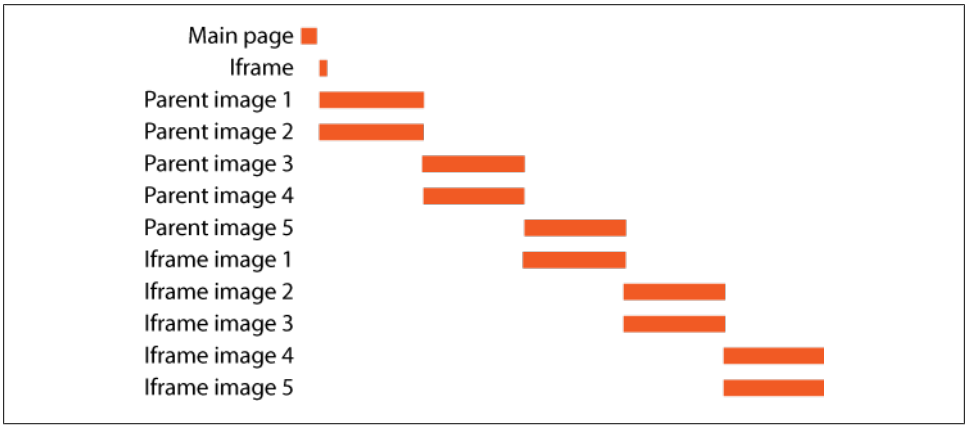


Figure 13-5. Iframe Connections HTTP waterfall chart for Internet Explorer 7

The first two requests are for the parent HTML document and iframe, respectively. The remaining requests are for the 10 images served from *1.cuzillion.com*. Internet Explorer 7 opens two connections per hostname. We see in Figure 13-5 that this limited pool of only two connections is shared by all the requests in the parent document as well as the iframe. This is the case for all major browsers.

Using an iframe does not increase the number of parallel downloads for a given hostname.

Connection Sharing Across Tabs and Windows

It's both surprising and disappointing that the connection limits apply across an iframe and its parent. This raises the question: is the connection pool similarly limited across multiple browser tabs and windows?

To answer this question, I created two URLs:

http://stevesouders.com/efws/connections1.php

http://stevesouders.com/efws/connections2.php

Similar to the previous example, each of these pages contains 5 images served from *1.cuzillion.com*, for a total of 10 images. The test involves opening two tabs in a browser and loading the URLs simultaneously. If the connection pool is shared, the 10 images

will take longer to download. If each browser tab has its own connection pool, the images will download in parallel and the overall load time will be faster. The same test is done using two instances of the same browser loading the URLs in separate windows.

I ran these tests on Internet Explorer 8.0 beta 2, Firefox 3.1b2, Safari 4 developer preview, Chrome 2.0, and Opera 10.0 alpha. The results are that the connection pool is shared across tabs and windows for all these browsers. Figure 13-6 shows the HTTP waterfall chart for Internet Explorer 8.0 beta 2.

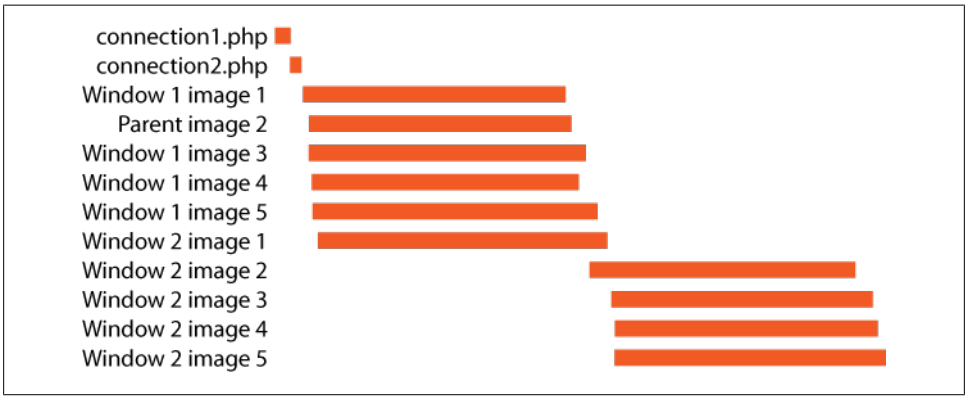


Figure 13-6. Connections shared across windows in Internet Explorer 8

The two test URLs are the first two short requests. The longer requests are the images. Internet Explorer 8 opens a maximum of six connections per server. This pool of six connections is used to download the five images in *connections1.php* and the first image in *connections2.php*. At that point, the remaining images in *connections2.php* are blocked, even though they're being fetched in a separate window.

This section is a digression from the main topic of iframes, but it's worth noting this behavior of connection limits being applied across tabs and windows. For companies that host multiple properties on a single domain, this could have a negative impact on performance if users open multiple web applications simultaneously. For example, several Google applications are hosted from *http://www.google.com*:

- Google Calendar
- Google Finance
- Google Reader
- Google Search
- iGoogle

Most of the resources used by these web sites also come from *http://www.google.com*. If a user opened two or more simultaneously, he would compete for the open connections, resulting in slower load times. Although this doesn't happen frequently, it does happen. For example, I have a script that opens Google Calendar, Google Reader, and

iGoogle every morning when I start my browser. These web sites are impacted because they must share connections even though they are loaded in separate tabs.

Summarizing the Cost of Iframes

Even blank iframes are expensive. They are one to two orders of magnitude more expensive than other DOM elements.

When used in the typical way (`<iframe src="url"></iframe>`), iframes block the `onload` event. This prolongs the browser's busy indicators, resulting in a page that is perceived to be slower. Setting the iframe's `SRC` dynamically avoids this problem in Safari and Chrome. For other browsers, setting the `SRC` after the `onload` event avoids the problem.

Although iframes don't directly block resource downloads in the main page, there are ways that the main page can block the iframe's downloads. In addition to the expected behavior of scripts, stylesheet downloads in the main page block the iframe's downloads in both Internet Explorer and Firefox.

The browser's limited connections per server are shared across the main page and iframes, even though an iframe is an entirely independent document. Web sites that host most of their resources on a single domain should keep this in mind.

With all of these costs, it's often best to avoid the use of iframes, and yet a quick survey shows that they are still used frequently. Five of the top 10 U.S. web sites use iframes: AOL, Facebook, MSN.com, MySpace, and YouTube. These sites use iframes primarily for serving ads. This is to be expected, given that iframes are an easy way to include content from a third-party site, especially dynamic content such as a rotating ad.

An alternative way to insert ads with better performance would be for the main page to create a `DIV` to hold the contents of the ad. When the main page requests the ad's external script (using an asynchronous technique as described in Chapter 4), the ID of this `DIV` could be included in the script's URL. The ad's JavaScript would then insert the ad in the page by setting the `innerHTML` of the `DIV`. This approach is also more compatible with "expando" ads—those ads that take over a large part of the window and thus cannot be constrained by an iframe. The use of iframes is declining as these other techniques for inserting ads become more prevalent, much to the benefit of web page performance.

CHAPTER 14

Simplifying CSS Selectors

Much of this book focuses on JavaScript performance. What about CSS? Most published information about CSS understandably focuses on layout, design, and the relationship between content, markup, and code.* There are a few best practices focused on CSS performance:

- Place stylesheets in the `HEAD` of the document to promote progressive rendering. (See *High Performance Web Sites*, Chapter 5.)
- Don't use CSS expressions in Internet Explorer, as they may be executed thousands of times, resulting in a sluggish page. (See *High Performance Web Sites*, Chapter 7.)
- Avoid too much inline styling as it increases download size. (See Chapter 9 of this book.)

Another topic that has garnered attention is the cost of inefficient CSS selectors. A *selector* is the initial list of arguments that indicates the elements of the page to which a CSS rule applies. This chapter explains the issues with regard to CSS selectors. There are some surprises. Although following the guidelines for optimal CSS selectors can make a difference in web site speed, it's more important that web developers avoid a few common, yet costly, CSS selector patterns. All of this is revealed in the sections that follow.

Types of Selectors

This section defines the terminology around CSS selectors. Consider this example:

```
#toc > LI { font-weight: bold; }
```

This is a style rule or simply *rule*. The *CSS selector* is `#toc > LI`. This selector contains two *simple selectors* (`#toc` and `LI`) that are joined with the `>` *combinator*. The CSS selector determines which elements in the page, also called *subjects*, receive the specified styling.

* See Nicole Sullivan's "Object Oriented CSS" and Nate Koechley's "Semantic Markup—Create, Support and Extract".

The browser tries to match the CSS selectors with elements in the document. It's this matching that is the cause for concern. The amount of matching the browser must perform depends on how the CSS selectors are written. Some types of CSS selectors cause more matching attempts and are thus more expensive than simpler selectors.

The various types of CSS selectors are presented in the following sections. They're listed in approximate order from simplest (least costly) to most complex (most expensive). For more information, consult the section in the CSS2 specification on selectors.

The example rules from this section are used in the CSS Selectors example. The use case for this example is styling for a table of contents. The page is shown in Figure 14-1 with some of the applicable rules indicating where they have an effect.

CSS Selectors
http://stevesouders.com/efws/selectors.php

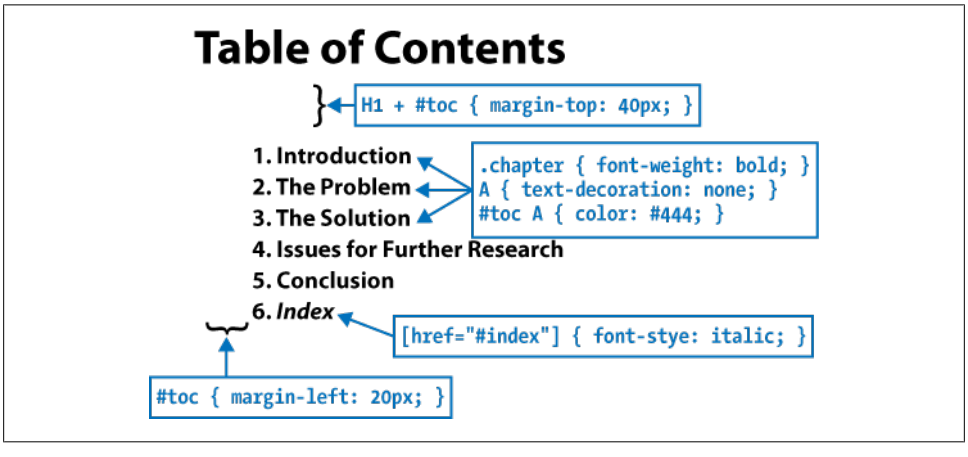


Figure 14-1. CSS Selectors example with rules

ID Selectors

Example: `#toc { margin-left: 20px; }`

Simple and efficient, this type of selector matches the unique element in the page with the specified ID. The example just shown matches the element whose ID attribute is `toc`. In the Table of Contents example, this rule matches an ordered list element: `<ol id=toc>`. This rule indents the list by 20 pixels on the left.

Class Selectors

Example: `.chapter { font-weight: bold; }`

Rules based on a class are specified with a dot (`.`) followed by the class name. Class selectors match all elements with a class attribute containing that name. This rule matches the list item elements in our Table of Contents example, making the font bold: `<li class=chapter>`.

Type Selectors

Example: `A { text-decoration: none; }`

Type selectors apply to all elements of the specified element type. This rule removes the underline from all anchors in the page, such as: `<a href="#introduction"> Introduction</a>`. This is a lightweight way to add styling to all elements of a specified type, without having to add any extra characters (such as ID, class, or style) to each element.

Adjacent Sibling Selectors

Example: `H1 + #toc { margin-top: 40px; }`

Adjacent sibling selectors are created by chaining two simple selectors (in this case, `H1` and `#toc`) with the `+` combinator. In our CSS Selectors page, this rule matches the `toc` element because its previous sibling is an `H1` element. As a result, the Table of Contents example is given an extra 40 pixels of margin at the top.

Child Selectors

Example: `#toc > LI { font-weight: bold; }`

Child selectors are formed by combining two or more simple selectors with the `>` combinator. This rule matches all list items whose parent is the `toc` element. This rule achieves the same results as the class selector example, but the `LI` elements don't need to specify a class, thus reducing the size of the resultant page.

Descendant Selectors

Example: `#toc A { color: #444; }`

The previous two selector types use the `+` and `>` combinators. Descendant selectors simply use a space (" ") as a combinator. Descendant rules are matched whenever the second selector subject is found to be a descendant (child, grandchild, etc.) of the first selector subject. In our page, all anchor (`A`) elements within the Table of Contents (`toc`) element are given a font color of "#444." Notice that I've used shorthand to

specify the color. Normally this color would be specified as "#444444," but "#444" means the same thing and saves three bytes.

Universal Selectors

Example: `* { font-family: Arial; }`

Universal selectors, written using *, match every element in the document. This rule assigns the Arial font to all elements in the CSS Selectors example page.

Attribute Selectors

Example: `[href="#index"] { font-style: italic; }`

Attribute selectors match based on the existence or value of an element's attributes. This rule causes the anchor element with `href` equal to `"#index"` to be drawn in italics. Attributes can be matched in four ways:

- Equality, using `=` as this rule does.
- Existence of the attribute, regardless of value: `[href]`.
- Equality with one value in a space-separated list of values: `[title~="Index"]` matches `<a title="the Index">`.
- Equality with the first value in a hyphen-separated list of values: `[LANG|=en]` matches `<p lang="en">` as well as `<p lang="en-US">`.

Class selectors are a specialized case of attribute selectors, where the attribute is `class`. The dot notation for class selectors (e.g., `.chapter`) is shorthand to avoid the lengthier attribute syntax (`[class="chapter"]`).

Pseudo-Classes and Pseudo-Elements

Example: `A:hover { text-decoration: underline; }`

The types of selectors presented so far have been based on information from the DOM. Some desired styling is not represented in the DOM. Pseudo-classes and pseudo-elements were created to address these situations. This rule draws an underline whenever the user hovers over an anchor; `:hover` is a pseudo-class. Other pseudo-classes include `:first-child`, `:link`, `:visited`, `:active`, `:focus`, and `:lang`. The pseudo-elements include `:first-line`, `:first-letter`, `:before`, and `:after`.

The Key to Efficient CSS Selectors

The impact of CSS selectors on performance derives from the amount of time it takes the browser to match the selectors against the elements in the document. Developers have some control over how long this matching takes by writing their selectors to be

more efficient. The path to efficient selectors starts by understanding how selector matching works.

Rightmost First

Consider the following rule:

```
#toc > LI { font-weight: bold; }
```

Most of us, especially those who read left to right, might assume that the browser matches this rule by moving from left to right, and thus, this rule doesn't seem too expensive. In our minds, we imagine the browser working like this: find the unique `toc` element and apply this styling to its immediate children who are `LI` elements. We know that there is only one `toc` element, and it has only a few `LI` children, so this CSS selector should be pretty efficient.

In reality, CSS selectors are matched by moving from right to left! With this knowledge, our rule that at first seemed efficient is revealed to be fairly expensive. The browser must iterate over every `LI` element in the page and determine whether its parent is `toc`.

Our descendant selector example is even worse:

```
#toc A { color: #444; }
```

Instead of just checking for anchor elements inside `toc`, as would happen if it was read left to right, the browser has to check every anchor in the entire document. And instead of just checking each anchor's parent, the browser has to climb the document tree looking for an ancestor with the ID `toc`. If the anchor being evaluated isn't a descendant of `toc`, the browser has to walk the tree of ancestors until it reaches the document root.

David Hyatt, Safari and WebKit architect, reveals this information in one of the most-referenced articles on CSS selector performance, "Writing Efficient CSS for use in the Mozilla UI":

> The style system matches a rule by starting with the rightmost selector and moving to the left through the rule's selectors. As long as your little subtree continues to check out, the style system will continue moving to the left until it either matches the rule or bails out because of a mismatch.

Writing Efficient CSS Selectors

Armed with the insight that selectors are matched right to left, we can take another look at our CSS selectors and tune them to be more efficient. Before we start, it would be nice to have some additional information, such as which CSS selectors are the most expensive, and some patterns for making it easier to fix them. Fortunately, David Hyatt's article provides guidelines for writing efficient selectors:

Avoid universal rules
: In addition to the traditional definition of universal selectors, Hyatt lumps adjacent sibling selectors, child selectors, descendant selectors, and attribute selectors into this category of "universal rules." He recommends using ID, class, and tag selectors exclusively.

Don't qualify ID selectors
: Because there is only one element in the page with a given ID, there's no need to add additional qualifiers. For example, `DIV #toc` is unnecessary and should be simplified to `#toc`.

Don't qualify class selectors
: Instead of qualifying class selectors for specific tags, extend the class name to be specific to the use case. For example, change `LI .chapter` to `.li-chapter`, or better yet, `.list-chapter`.

Make rules as specific as possible
: Don't be tempted to build long selectors, such as `OL LI A`. It's better to create a class, such as `.list-anchor`, and add it to the appropriate elements.

Avoid descendant selectors
: Descendant selectors are typically the most expensive to process. Child selectors are often what's intended and can be more efficient. It's even better to follow the next guideline to avoid child selectors as well.

Avoid tag-child selectors
: If you have a child selector that is based on a tag, such as `#toc > LI > A`, use a class associated with each of those tag elements, such as `.toc-anchor`.

Question all usages of the child selector
: This is another reminder to review all places where child selectors are used, and replace them with specific classes when possible.

Rely on inheritance
: Learn which properties are inherited, and avoid rules that specify these inherited styles. For example, specify `list-style-image` on the list element instead of on each list item element. Consult the list of inherited properties to know which properties are inherited for which elements.

It's interesting to note that David Hyatt's article was first published in April 2000. I wonder—why is there renewed interest in this topic nine years later? David's article, as the title states, was addressed to developers working on the Mozilla UI. Perhaps it's taken this long for web pages to reach a similar level of performance loss with regard to CSS selectors.

Another factor is that today's Web 2.0 applications have a longer session length—it's not the load-clear-load Web 1.0 scenario. In this sense, Web 2.0 applications are more similar to the Mozilla UI, and the impact of inefficient CSS selectors may be more pronounced as huge portions of the DOM tree are created and removed, and DHTML code changes class names and style attributes. The findings in the next section support

this view that the complexity and dynamic nature of web pages are what have brought focus to this area of performance analysis.

CSS Selector Performance

The veil has been lifted. We now see the inefficiencies in our CSS selectors. The revelation of selectors being read from right to left motivates many of us to rewrite our rules. With Doug Crockford's guidance (see Chapter 1), we know that before we start fixing this possible performance issue, it's important to first measure the impact of the issue so that we are sure to focus on the right problem.

Complex Selectors Impact Performance (Sometimes)

The results of an experiment to measure the performance of CSS selectors were published in three blog posts from Jon Sykes. Each post is a refinement on the previous one, so part 3 is the most informative.† His test comprises five pages. All the pages contain 20,000 anchor elements, each with an ancestor tree of `P`, `DIV`, `DIV`, `DIV`, and `BODY`. Each page has different types of CSS:

- "No Style" has no CSS.
- "Tag" has one rule:

```
A { background-color: red; }
```

- "Class" has 20,000 class selectors, one for each anchor; for example:

```
class11 { background-color: red; }
```

- "Descender" has 20,000 descendant selectors, one for each anchor; for example:

```
div div div p a.class11 { background-color: red; }
```

- "Child" has 20,000 child selectors, one for each anchor; for example:

```
div > div > div > p > a.class11 { background-color: red; }
```

The results indeed show that "No Style" is faster than "Descender" and "Child." In Internet Explorer and Safari, the load times of the slower pages are a multiple of the simple page. It's clear that inefficient CSS selectors in numbers this large adversely affect performance.

What if the number of selectors is reduced to levels comparable to today's web sites—do they still have an impact? Table 14-1 shows the number of CSS rules and DOM elements, as well as average DOM depth, for the top 10 U.S. web sites. The total number of rules ranges from 92 to 2,882, with an average of 1,033.

† The original blog post, *http://blog.archive.jpsykes.com/153/more-css-performance-testing-pt-3/*, is no longer available.

Table 14-1. CSS rules and DOM elements in the top 10 U.S. web sites

Web site	Number of rules	Number of DOM elements	Average depth
AOL	2,289	1,628	13
eBay	305	588	14
Facebook	2,882	1,966	17
Google	92	552	8
Live Search	376	449	12
MSN.com	1,038	886	11
MySpace	932	444	9
Wikipedia	795	1,333	10
Yahoo!	800	564	13
YouTube	821	817	9
Average	**1,033**	**923**	**12**

Based on this information, I created a set of tests similar to Sykes' experiments, but instead of 20,000 rules they contain only 1,000 rules. Also, to make the page size more consistent, I gave the baseline page and tag selector page 1,000 rules just like all the other pages; these are simple class rules that aren't used by any elements. The pages themselves are part of the CSS Selectors Test example.

CSS Selectors Test
http://stevesouders.com/efws/css-selectors/tests.php

The focus of this experiment is to gauge the cost of complex selectors versus simple selectors. Figure 14-2 shows the difference in load times of the slowest test page (child or descendant selectors) and the simple baseline page. The average slowdown is just 30 milliseconds.‡

These tests show a much smaller savings from optimizing CSS selectors than what was found in Sykes' tests. This is primarily due to the reduced number of rules coupled with the fact that the impact of CSS selectors increases at a nonlinear rate as the number of rules and DOM elements grows. Figure 14-3 shows the page load time in Internet Explorer 7 as the number of rules increases from 1,000 to 20,000 for the more expensive child and descendant selector tests. This data reveals that Internet Explorer 7 hits a cliff at around 18,000 rules. The tests with 20,000 rules are on the extreme end of this hockey stick.

‡ The results for Opera 9.63 were inconsistent and so are omitted.

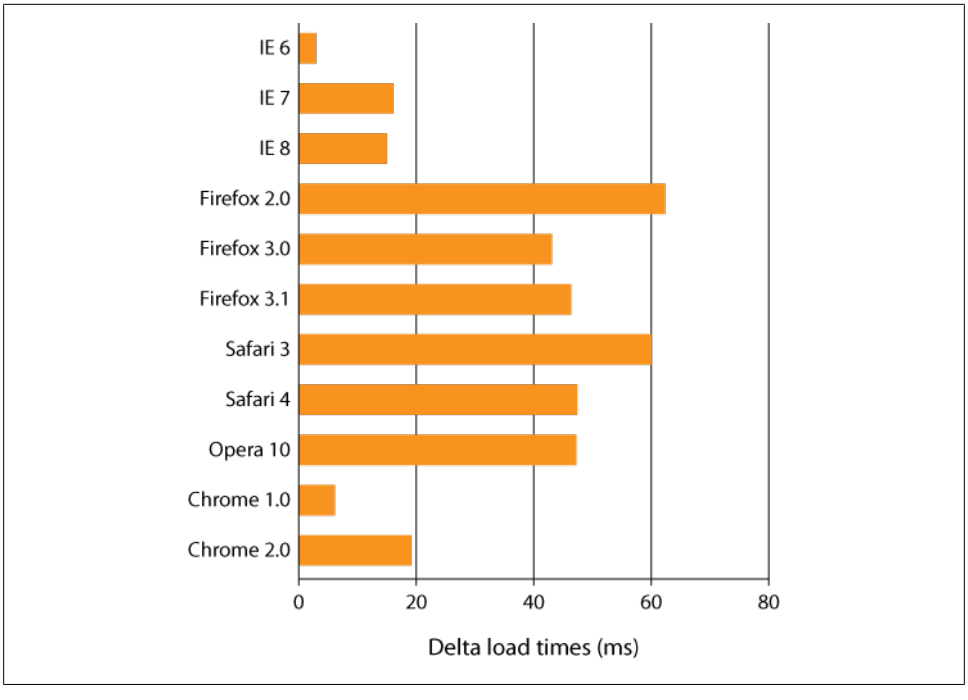


Figure 14-2. Load time difference for simple versus complex selector tests

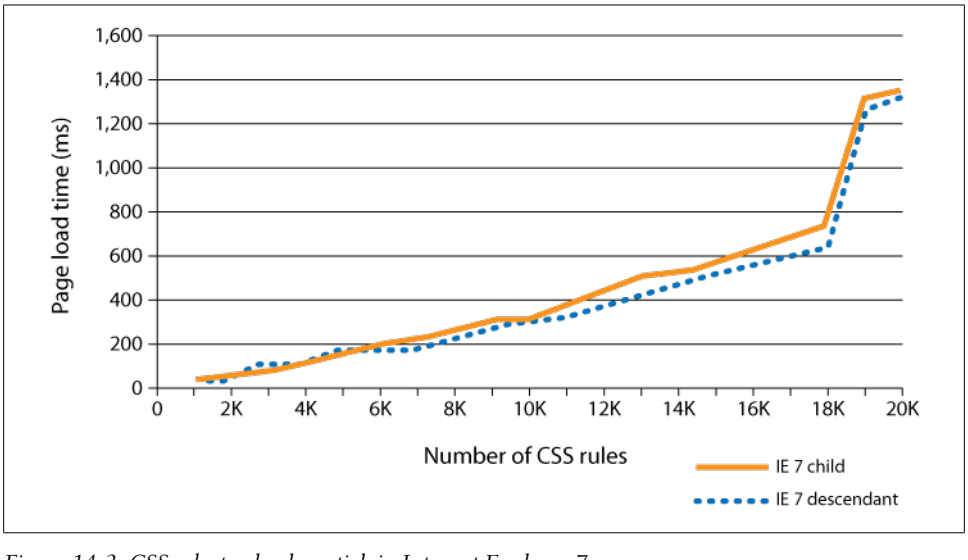


Figure 14-3. CSS selector hockey stick in Internet Explorer 7

These results indicate that more complex CSS selectors, such as child and descendant selectors, don't always affect page performance. This doesn't mean we shouldn't optimize our CSS selectors. Even at real-world levels, certain types of selectors have a noticeable impact on performance.

CSS Selectors to Avoid

The tests from the previous section show that in some situations, even complex CSS selectors have little impact on performance, but that's not always the case. Let's look at a sample descendant selector from the earlier tests:

```
DIV DIV DIV P A.class0007 { ... }
```

At first glance, this seems likely to be an expensive selector to match. It's a descendant selector, with five levels of ancestors to match. However, recalling that selectors are matched right to left, we realize why this descendant selector performs at about the same speed as a much simpler class selector. The amount of work performed by the browser is heavily affected by the rightmost argument, also called the *key selector*. In this case, the key selector is `A.class0007`. Only one element in the page matches this key selector, so the amount of time needed to match this selector is minimal.

In contrast, consider this rule:

```
A.class0007 * { ... }
```

In this rule, the key selector is *. Since this matches all elements, the browser has to check every element to see whether it is a descendant of an anchor with the class name `class0007`. The Universal Selector example has 1,000 rules of this type.

Universal Selector
: *http://stevesouders.com/efws/css-selectors/universal.php*

Figure 14-4 shows the difference in load times of the Universal Selector page and the Descendant Selector page. This is a significant change from Figure 14-2, where the average delta was just 30 milliseconds. The average slowdown here is more than two seconds!

When deciding where to optimize, remember to focus on CSS selectors where the *key selector* is likely to match a large number of elements in the page. It's not just the universal selector that is troublesome. Here are some other examples that take significant time to load:

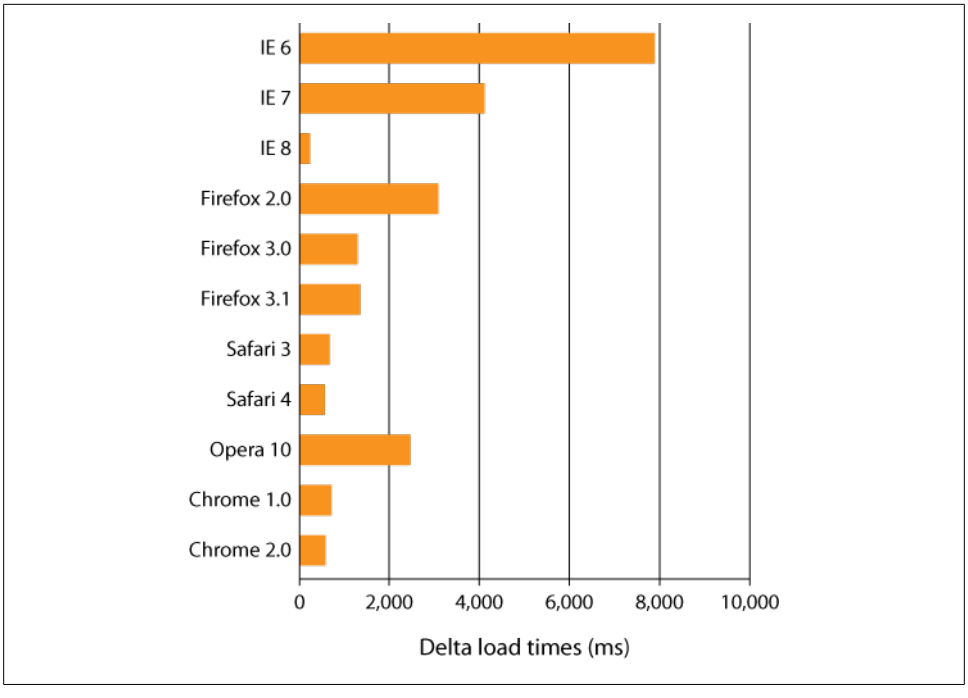


Figure 14-4. Load time difference for universal selector

`A.class0007 DIV { ... }`
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=A.class+DIV

`#id0007 > A { ... }`
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=%23id+>+A

`.class0007 [href] { ... }`
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=.class+[href]

`DIV:first-child { ... }`
http://stevesouders.com/efws/css-selectors/csscreate.php?sel=DIV%3Afirst-child

These examples are all created by the CSS Test Creator page. "0007" is used to indicate a counter that is incremented from 1 to the maximum number of rules (1,000 in this case). The CSS Test Creator makes it easy to try different types of selectors and measure the impact on load times, as well as reflow time, as discussed in the next section.

Reflow Time

All the examples so far have measured the impact of CSS selectors on load time. For Web 2.0 applications, a more important consideration is the time it takes the browser to apply styles and lay out elements while the user interacts with the page. This is called the *reflow time*. A reflow is triggered when certain properties of a DOM element's style

are modified using JavaScript. Given a DOM element called `elem`, each of the following lines of code triggers a reflow in most browsers:

```
elem.className = "newclass";
elem.style.cssText = "color: red";
elem.style.padding = "8px";
elem.style.display = "";
```

This is just a subset; the list of reflow triggers is much longer. Given their dynamic nature, Web 2.0 applications can easily trigger a reflow. A reflow doesn't necessarily involve every element in the page. Browsers are optimized to re-layout only the elements that are affected ("dirty"). In the preceding examples, however, if `elem` is the document body or some other element with many descendants, the reflow can be costly.

A reflow requires that CSS rules be reapplied, which means the browser must once again match all the CSS selectors. If the CSS selectors are inefficient, the reflow may take a long time—long enough that users notice. All of the CSS selector test examples have a Measure Reflow button. When clicked, the body's display property is toggled, as shown in the last line of the preceding code. The time it takes for the reflow to finish is displayed next to the button. The examples of expensive CSS selectors from the previous section have reflow times ranging from hundreds of milliseconds to seconds.

It's important, therefore, to be wary of the impact of inefficient CSS selectors not only on page load time, but also on how your Web 2.0 application behaves while the user interacts with it. If your JavaScript manipulates style properties and your page starts to feel sluggish, inefficient CSS selectors might be the cause.

Measuring CSS Selectors in the Real World

This chapter presents the results from multiple experiments, but all of these test pages are contrived examples. It's difficult to translate the time savings shown by these tests into savings for real-world web sites. An ideal experiment would be to optimize the CSS selectors in the top 10 web sites and measure the effect on load time, but that's not feasible.

To estimate the performance improvement that might be gained by optimizing CSS selectors, we can measure the reflow time. This is easy to do on existing web pages using Lindsey Simon's Reflow Timer. This is a bookmarklet that runs in all major browsers. When launched, it toggles the body's display property and displays the average reflow time. (I got the idea to add reflow time measurements to my test pages based on this tool.) Figure 14-5 shows the results of measuring reflow time. Reflow time ranges from 16 milliseconds (Google Search) to 391 milliseconds (Facebook).[§]

[§] Measured using Internet Explorer 7.

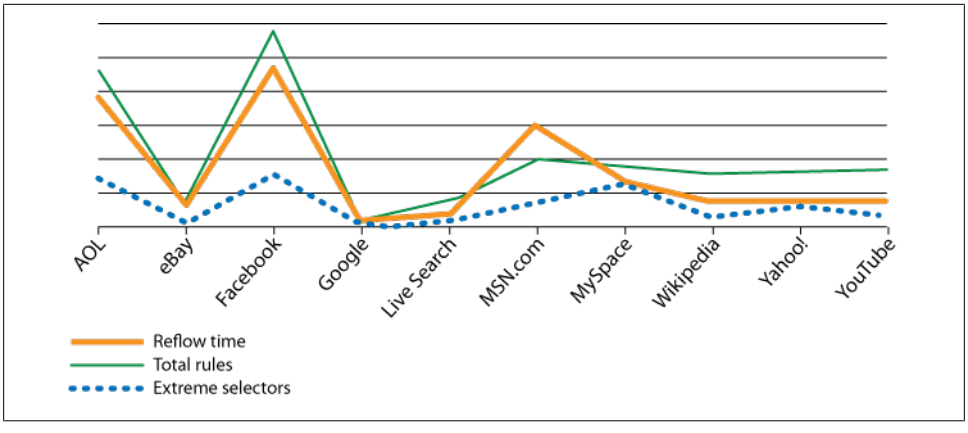


Figure 14-5. Top 10 sites reflow time, total rules, and extreme selectors

In addition to reflow time, Figure 14-5 also shows the number of rules and the number of extremely inefficient rules (rules with key selectors that match a large number of elements). The correlation coefficient between reflow time and the number of rules is 0.86. The correlation coefficient between reflow time and the number of extremely inefficient rules is 0.9. Both of these are high correlations, suggesting that the time it takes for a browser to apply styles is affected by both the number as well as the efficiency of CSS selectors.

If, like AOL and Facebook, your site use a large number of rules, many of which are extremely inefficient, optimizing your CSS selectors may make your page faster. You'd likely also benefit from reducing the number of rules. Having said that, keep in mind that there's a cost associated with following David Hyatt's guidelines for writing efficient CSS selectors: replacing expensive descendant selectors with class assignments for each affected element adds weight to your page and reduces the flexibility of your styles. The most important selectors to fix are those with a key selector (rightmost selector) that matches a large number of elements. Although the benefit of this performance improvement varies, web developers should be aware that certain types of CSS selectors can torpedo their page's performance.

APPENDIX

Performance Tools

Like all good engineers, web developers need to build up a set of tools to do a high-quality job. This appendix describes the tools I recommend for analyzing and improving web site performance. The tools are grouped into the following sections:

"Packet Sniffers" on page 205
When I sit down to analyze a web site, I start by looking at the HTTP requests for the web page in question. This makes it possible to identify the slow parts of the page. A packet sniffer that is handy and easy to use is the first tool to add to your kit. The tools included in this category are HttpWatch, Firebug Net Panel, AOL Pagetest, VRTA, IBM Page Detailer, Web Inspector Resources Panel, Fiddler, Charles, and Wireshark.

"Web Development Tools" on page 209
Page performance isn't just about load time—JavaScript, CSS, and DOM structure play a significant role, especially for Web 2.0 applications. Web development tools provide inspectors, profilers, and debuggers for analyzing web page behavior. This section includes Firebug, Web Inspector, and IE Developer Toolbar.

"Performance Analyzers" on page 211
Performance analyzers evaluate a given web page against a set of performance best practices. As I will explain, they vary a great deal in what they measure. This section describes YSlow, AOL Pagetest, VRTA, and neXpert.

"Miscellaneous" on page 216
This section includes a grab bag of tools I use regularly: Hammerhead, Smush.it, Cuzillion, and UA Profiler.

Packet Sniffers

Every web developer working on performance needs to look at how his web pages load, including all the resources within the page. This is done using a *packet sniffer*. The packet sniffers listed in this section range from tools that give a higher-level view of network traffic, such as HttpWatch, to lower-level tools that expose each packet sent

over the network, such as Wireshark. In most of my web performance analysis, I use the higher-level network monitors; they are typically easier to configure and have a user interface that is more visual. In some situations, such as debugging chunked encoding, I drop down into one of the lower-level packet sniffers in order to see the contents and timing of each packet sent over the wire.

HttpWatch

HttpWatch is my preferred packet sniffer. HttpWatch depicts network traffic in a graphical way, as shown in Figure A-1. Most of the HTTP waterfall charts in this book were captured using HttpWatch. This graphical display makes it easy to spot performance delays.

HttpWatch is built by Simtec. You can try out the free download, but it's limited to work on only a few major sites, such as Google and Yahoo!. You have to pay for the full-featured version, but it's money well spent. HttpWatch runs on Microsoft Windows with Internet Explorer and Firefox.

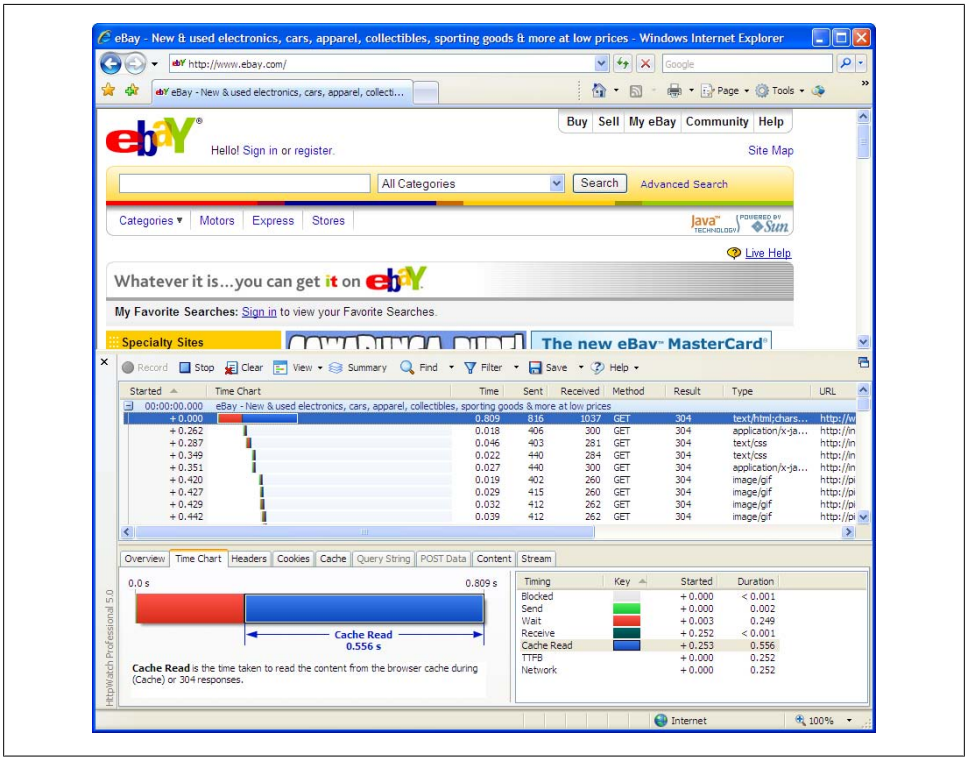


Figure A-1. HttpWatch

Firebug Net Panel

Firebug has many features critical to any web developer and is described more thoroughly in "Web Development Tools" on page 209. The Firebug Net Panel, however, deserves mention here. Net Panel displays HTTP waterfall charts, making it an easy alternative for developers who already have Firebug installed. I especially like Net Panel's use of vertical lines to mark the `DOMContentLoaded` and `onload` events in the page load timeline, as shown in Figure A-2. This is a feature that other packet sniffers should adopt.

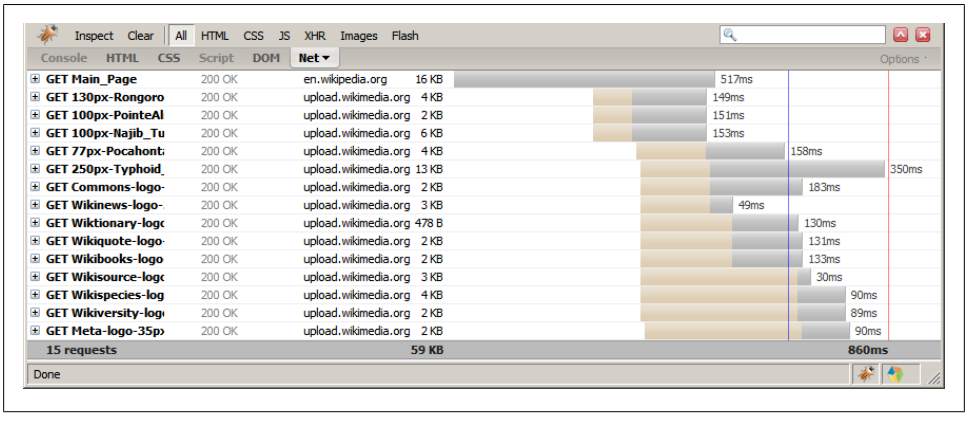


Figure A-2. Firebug Net Panel

One drawback of Net Panel is that the timing information can be affected by the web page itself. This is due to the fact that Firebug is implemented in JavaScript and therefore executes in the same Firefox process as the current web page. Because of this, if network events happen while JavaScript is executing in the main page, Net Panel is blocked from recording the correct timing information for those requests. Net Panel's accuracy is sufficient in most situations, and its ease of use makes it a good choice. If you require more precise time measurements or have a page with long-running blocks of JavaScript, you should consider using one of the other packet sniffers mentioned in this section.

An additional constraint is that Firebug is a Firefox add-on, so it isn't available in other browsers.

AOL Pagetest

AOL Pagetest is an Internet Explorer plug-in that produces HTTP waterfall charts. It also identifies areas for improving performance, as discussed in "Performance Analyzers" on page 211.

VRTA

VRTA from Microsoft focuses on improving network performance. Its HTTP waterfall chart is more detailed than other network monitors, putting an emphasis on reusing existing TCP connections. See "Performance Analyzers" on page 211 for more information about VRTA.

IBM Page Detailer

IBM Page Detailer used to be my preferred packet sniffer, but IBM stopped selling the professional version. The basic version is still available, but it lacks many features that I consider mandatory, such as support for analyzing HTTPS requests and the ability to export data. IBM Page Detailer runs on Microsoft Windows.

I use IBM Page Detailer when analyzing browsers other than Internet Explorer and Firefox, such as Opera and Safari (since these browsers aren't supported by HttpWatch). IBM Page Detailer can monitor network traffic for any process that uses HTTP. This is enabled by editing the *wd_WS2s.ini* file and adding the process's name to the `Executable` line, like so:

```
Executable=(FIREFOX.EXE),(OPERA.EXE),(SAFARI.EXE)
```

There's an interesting twist that prevents IBM Page Detailer from analyzing Chrome: Chrome has a separate process for the browser UI plus one for each tab. IBM Page Detailer attaches to the browser UI process, and so it doesn't see any of the HTTP traffic for the actual web pages being loaded. Nevertheless, if support for HTTPS and exporting data isn't required, IBM Page Detailer is a useful alternative.

Web Inspector Resources Panel

Safari's Web Inspector, similar to Firebug, is a web development tool that includes a network monitor. See "Web Development Tools" on page 209 for more information.

Fiddler

The main distinguishing feature of Fiddler, built by Eric Lawrence from the Microsoft Internet Explorer team, is that it supports a scripting capability that allows for setting breakpoints and manipulating HTTP traffic. One downside is that it acts as a proxy, and so it may alter the behavior of the browser (e.g., the number of open connections per server). If you need a scripting capability and are mindful of any side effects of using a proxy, I highly recommend Fiddler. It runs on Microsoft Windows.

Charles

Charles is an HTTP proxy, similar to Fiddler. It has many of the same features as Fiddler, including the ability to analyze both HTTP and HTTPS traffic, and bandwidth throttling. Charles supports Microsoft Windows, Mac OS X, and Linux.

Wireshark

Wireshark evolved from Ethereal. It analyzes HTTP requests at the packet level. Its UI is not as graphical as other network monitors. It also doesn't have the concept of a "web page," so it's up to you to discern where the web page's packets start and end. If you have to look at traffic at the packet level, such as to analyze chunked encoding, Wireshark is the best choice. It's available on many platforms, including Microsoft Windows, Mac OS X, and Linux.

Web Development Tools

Packet sniffers show the network activity while a page is loading, but there's more to a web page's performance than just HTTP requests. Chapters 1 and 2 discuss how JavaScript and modifications to the DOM can slow a page down. The web development tools presented in this section—Firebug, Web Inspector, and IE Developer Toolbar—include features such as DOM inspectors, JavaScript debuggers and profilers, CSS editors, and network monitors.

These tools are the tip of the iceberg. More extensive tools are needed to give developers visibility into memory consumption, CPU load, JavaScript execution, CSS application, and HTML parsing and rendering over the entire page load timeline. And this analysis is needed without altering normal browser behavior.

Firebug

Firebug is the most popular web development tool, with more than 14 million (yes, million!) downloads. It was created by Joe Hewitt in January 2006. It includes inspectors for HTML, CSS, DOM, and layout. Firebug's Net Panel, discussed in "Packet Sniffers" on page 205, provides an HTTP waterfall chart of network activity. Firebug also has a JavaScript command line and console, as well as a JavaScript debugger and profiler. The debugger and profiler are Firebug's strongest features.

Firebug is an add-on to Firefox. Although porting the JavaScript debugging and profiling functionality to other browsers would be a tremendous undertaking, many of Firebug's other features are available across browsers by virtue of Firebug Lite. Firebug Lite is a bookmarklet, and therefore it works in all the major browsers. It had a major upgrade by Azer Koçulu and now includes inspectors for HTML, DOM, and CSS, as well as a JavaScript command line and console. Providing a common UI across all

browsers and a fairly complete set of features, Firebug Lite is the perfect recipe for solving nasty browser incompatibility bugs.

Developers love Firebug because of their ability to extend it. This open extension model makes it possible to add on to Firebug's features in a way that also allows for that new functionality to be shared with other developers. You can find useful Firebug extensions at *http://getfirebug.com/extensions/index.html*.

Web Inspector

Safari's Web Inspector had a significant upgrade at the end of 2008. The Resources Panel, mentioned previously, is shown in Figure A-3. Web Inspector's functionality is similar to Firebug. It has a console with autocompletion, a DOM and CSS inspector, and a JavaScript debugger and profiler.

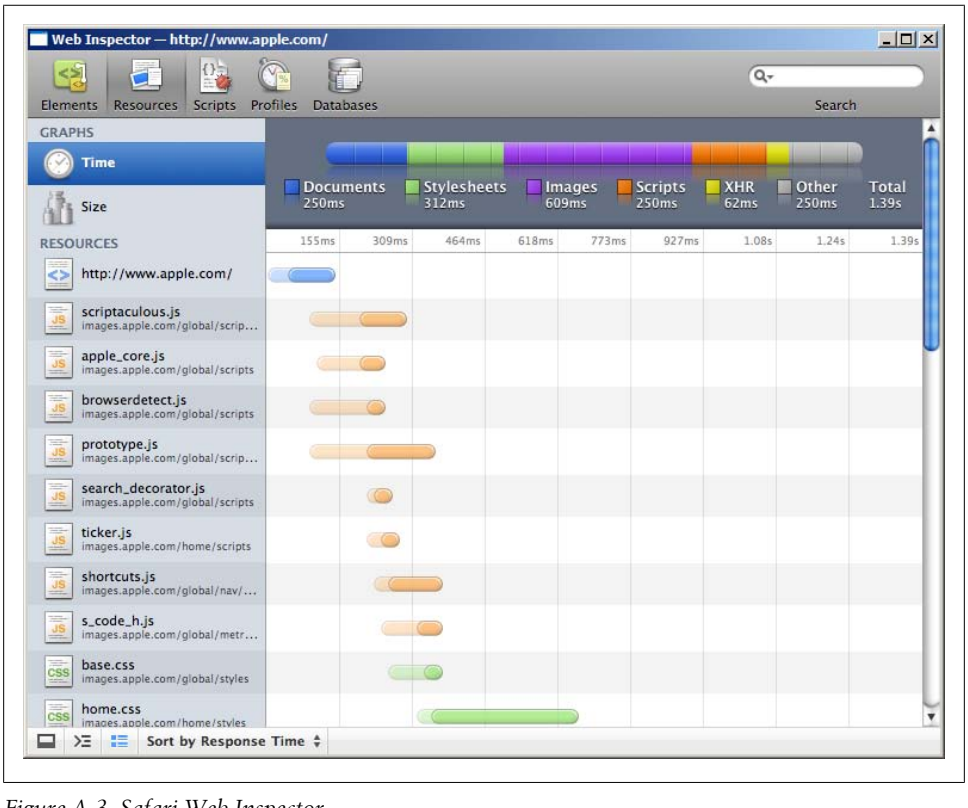


Figure A-3. Safari Web Inspector

IE Developer Toolbar

The Internet Explorer Developer Toolbar has a feature set similar to Firebug Lite. It doesn't have JavaScript debugging or profiling, but it does support validating HTML and CSS, DOM inspection, and pixel layout tools. The IE Developer Toolbar is targeted at Internet Explorer 6 and 7. The functionality has been built into Internet Explorer 8 under the Developer Tools menu item.

Performance Analyzers

YSlow was the first widely used performance "lint" tool. AOL Pagetest, VRTA, and neXpert were released subsequently. Each of these tools has its own set of performance best practices. I've aggregated all of these best practices in Table A-1, with an indication of which rules are evaluated by each particular tool. I've grouped the best practices into three categories:

- The rules included in *High Performance Web Sites*
- The best practices described in this book
- Other rules that I haven't addressed but that are incorporated in at least one of these tools

Looking at Table A-1, it's clear that there is little overlap in the best practices espoused by each tool. In one sense, this is good—bringing in different perspectives on the performance problem leads to the discovery of new best practices. But this diversity has a more important and unfavorable impact: confusion and fragmentation in the web development community. It's unclear which set of best practices is best. The choice of tool might be dictated by development environment rather than by the content of the performance analysis.

Across the developers of these tools, there is more agreement on performance best practices than is reflected in Table A-1. The inconsistencies arise for several reasons. There's a desire to introduce new best practices and to focus less on covering what has already been covered somewhere else. Development time is always an issue; developers may decide to skip the implementation of well-known best practices. Don't underestimate the impact of personal interests; for instance, it's clear that the developers of VRTA have more interest and familiarity with networking issues than I do.

Table A-1. Performance best practices

Best practice	YSlow	Pagetest	VRTA	neXpert
High Performance Web Sites				
Combine JavaScript and CSS	X	X		
Use CSS sprites	X		X	
Use a CDN	X	X		
Set `Expires` in the future	X	X	X	X
Gzip text responses	X	X	X	X
Put CSS at the top	X			
Put JavaScript at the bottom	X			
Avoid CSS expressions	X			
Make JavaScript and CSS external	X			
Reduce DNS lookups	X			
Minify JavaScript	X	X		
Avoid redirects	X		X	X
Remove dupe scripts	X			
Remove ETags	X	X		X
Even Faster Web Sites				
Don't block the UI thread				
Split JavaScript payload				
Load scripts asynchronously			X	
Inline scripts before stylesheet				
Write efficient JavaScript				
Minimize uncompressed size				
Optimize images		X		
Shard domains			X	
Flush the document early				
Avoid iframes				
Simplify CSS selectors			X	
Other				
Use persistent connections		X	X	X
Reduce cookies		X		X
Avoid network congestion			X	
Increase MTU, TCP window			X	
Avoid server congestion			X	

Moving forward, web developers would be well served if it became possible for these and other tools to share a common set of performance best practices. I fully expect this will happen. These tools were created in the spirit of evangelizing a faster web experience for all users and to help developers easily identify where they can make the greatest improvement to their site's speed. In that spirit, it makes sense to give developers tools that are more consistent regardless of their platform and tool of choice.

That's the future. For now, the following sections provide descriptions of YSlow, AOL Pagetest, VRTA, and neXpert, as they exist today.

YSlow

I created YSlow while working at Yahoo!. It existed first as a bookmarklet, and then as a Greasemonkey script. Joe Hewitt was kind enough to explain how to port YSlow to be a Firebug extension. Swapnil Shinde did a lot of the coding to get it to work with Firebug. The motivation I gave Swapnil was that I was certain YSlow would be used by as many as 10,000 people. YSlow was released in July 2007 and crossed the 1 million download mark a year and a half later. The name is a play on the question "wh*Y* is this page *Slow*?"

YSlow contains the following rules which are echoed as chapters in *High Performance Web Sites*. When YSlow was released, I also posted summaries of each rule at *http://developer.yahoo.com/performance/rules.html*. That page has subsequently been updated by the folks at Yahoo! to include 34 rules! Here are the original 13 rules that are still the basis for YSlow's performance analysis:

- Rule 1: Make Fewer HTTP Requests
- Rule 2: Use a Content Delivery Network
- Rule 3: Add an `Expires` Header
- Rule 4: Gzip Components
- Rule 5: Put Stylesheets at the Top
- Rule 6: Put Scripts at the Bottom
- Rule 7: Avoid CSS Expressions
- Rule 8: Make JavaScript and CSS External
- Rule 9: Reduce DNS Lookups
- Rule 10: Minify JavaScript
- Rule 11: Avoid Redirects
- Rule 12: Remove Duplicate Scripts
- Rule 13: Configure ETags

YSlow, as an extension to Firebug, is available only within Firefox. It generates a score for each rule and an overall score based on a weighted average of the individual rule scores. It also displays a list of all the resources used in the page as well as overall

statistics (number of requests, total page weight, etc.). It has other useful tools, including integration with JSLint and output of all the CSS and JavaScript into a single browser window for easy searching.

AOL Pagetest

AOL Pagetest and its web-based counterpart, WebPagetest, analyze web pages using these best practices:

- Enable browser caching of static assets
- Use one CDN for all static assets
- Combine static CSS and JavaScript files
- Gzip-encode all appropriate text assets
- Compress images
- Use persistent connections
- Proper cookie usage
- Minify JavaScript
- No ETag headers

AOL Pagetest is a plug-in for Internet Explorer. WebPagetest is accessible through any browser; it runs Internet Explorer on the backend server. In addition to performance analysis, both provide an HTTP waterfall chart, screenshots, page load times, and summary statistics.

The deployment of this functionality via the WebPagetest web site is intriguing. WebPagetest is fairly popular, but it hasn't gotten the wide adoption it deserves. It lets you analyze any web site from any browser, without the hassle of downloading, installing, and configuring an application or plug-in. It does this by running AOL Pagetest in Internet Explorer on the WebPagetest site's backend servers. WebPagetest users, from any browser, simply enter the URL of the site they want to analyze into the web-based form, and the results are presented a minute or so later. Figure A-4 shows the results for *http://www.aol.com/*.

Making WebPagetest available through a web page form makes it easy to use for everyone, including nondevelopers, but it does have some limitations. It's important to remember that the results are always generated using Internet Explorer running in WebPagetest's remote location. This can be confusing. Notice in Figure A-4 that I'm using Firefox; remembering that these results were produced using Internet Explorer is a challenge. Similarly, the results do not necessarily reflect your local conditions. If you're trying to debug a problem with your current Internet connection, or you're loading a page that depends on your current cookies, that can't be captured by WebPagetest. AOL Pagetest (the downloaded, locally installed Internet Explorer plug-in) or the other packet sniffers mentioned in the previous section are the choice for analyzing your current browsing experience.

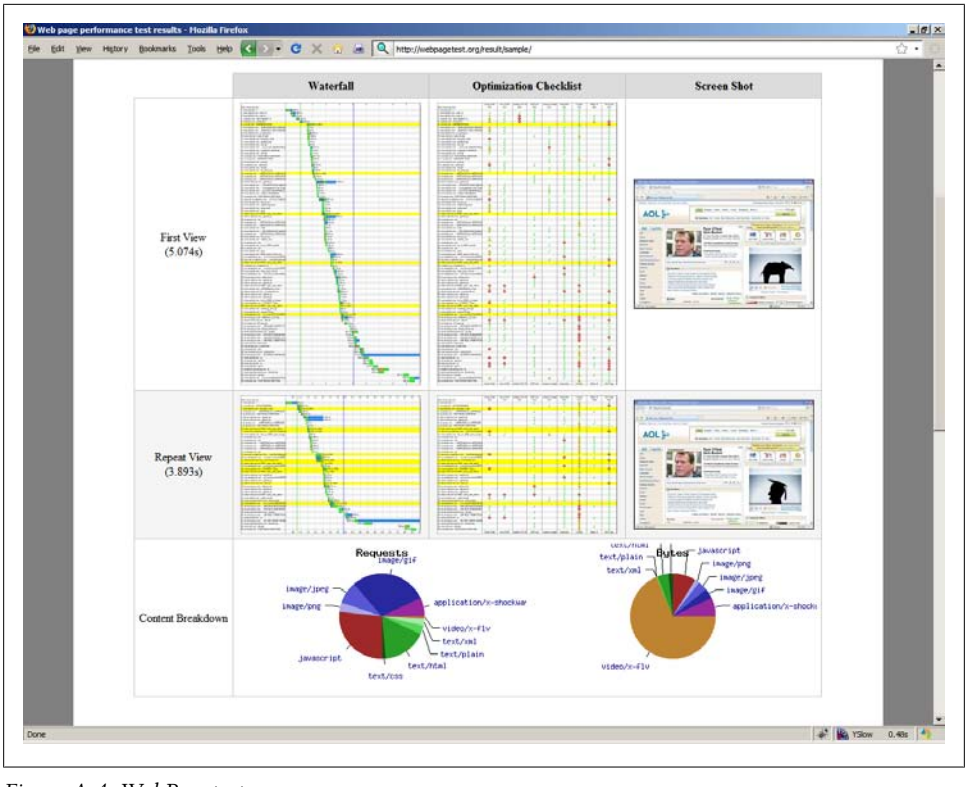


Figure A-4. WebPagetest

VRTA

VRTA from Microsoft is short for Visual Round Trip Analyzer. It displays HTTP waterfall charts, but these are more detailed than those found in other tools. VRTA focuses on network optimization. One key aspect of this is reusing existing TCP connections. In most HTTP waterfall charts, each HTTP request is a separate horizontal bar. Instead, VRTA represents each TCP connection as a horizontal bar. This makes it easy to see how well TCP connections are being utilized. VRTA also shows a bit rate histogram, to show how well the available bandwidth is utilized.

In addition to its sophisticated network charts, VRTA evaluates the page download information against the following set of performance best practices:

- Open enough ports
- Limit the number of small files to be downloaded
- Load JavaScript files outside of the JavaScript engine
- Turn on keepalives
- Identify network congestion
- Increase network maximum transmission unit (MTU) or TCP window size
- Identify server congestion
- Check for unnecessary round trips
- Set expiration dates
- Think before you redirect
- Use compression
- Edit your CSS

neXpert

neXpert is also from Microsoft. It's an add-on to Fiddler (see "Packet Sniffers" on page 205 for more information about Fiddler). It uses Fiddler to gather information about the resources downloaded for a web page. neXpert analyzes this information against a set of performance best practices and produces a report of suggested improvements. neXpert goes further than other performance analyzers in that it predicts the impact these improvements might have on the web page's load time. The list of performance best practices analyzed by neXpert includes the following:

- HTTP response codes
- Compression
- ETags
- Cache headers
- Connection header
- Cookies

Miscellaneous

The tools in this section address specific web performance areas not covered in the previous sections. I use all of these tools on a regular, if not daily, basis.

Hammerhead

Improving web performance requires measuring page load times. Although this sounds simple, in reality it's extremely difficult to gather load time measurements in an accurate and statistically sound way that is representative of real-world users. There's no single

solution. Instead, multiple techniques are required, including measuring real-world traffic, bucket testing, and scripted or synthetic testing. The problem is that all of these techniques are costly, in terms of both dollars and calendar time.

I created Hammerhead to make it easier for developers to measure load times early in the development process. Hammerhead is an extension to Firebug. To test, or "hammer," a set of web pages, enter the URLs into Hammerhead, along with the number of measurements desired. Figure A-5 shows an example.

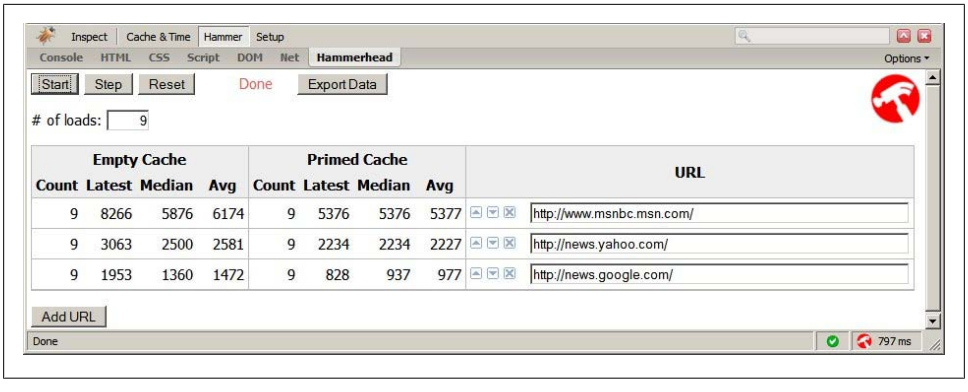


Figure A-5. Hammerhead

Hammerhead loads each URL the specified number of times and records each measurement, as well as the average and median load times. The pages are loaded with both an empty and a primed cache (Hammerhead manages the cache for you). Although Hammerhead measurements are gathered under just one set of test conditions (your development environment), they provide a quick and easy way to compare two or more web page alternatives.

Smush.it

Smush.it is a service for analyzing and optimizing images in your web page. It was created by Stoyan Stefanov and Nicole Sullivan, the authors of Chapter 10. Smush.it tells you how many bytes you can save by optimizing your images, as shown in Figure A-6. It even produces the optimized images for you as a single ZIP file for easy download. There is also a Smush.it bookmarklet and Firefox extension, so you can get similar functionality inside the browser.

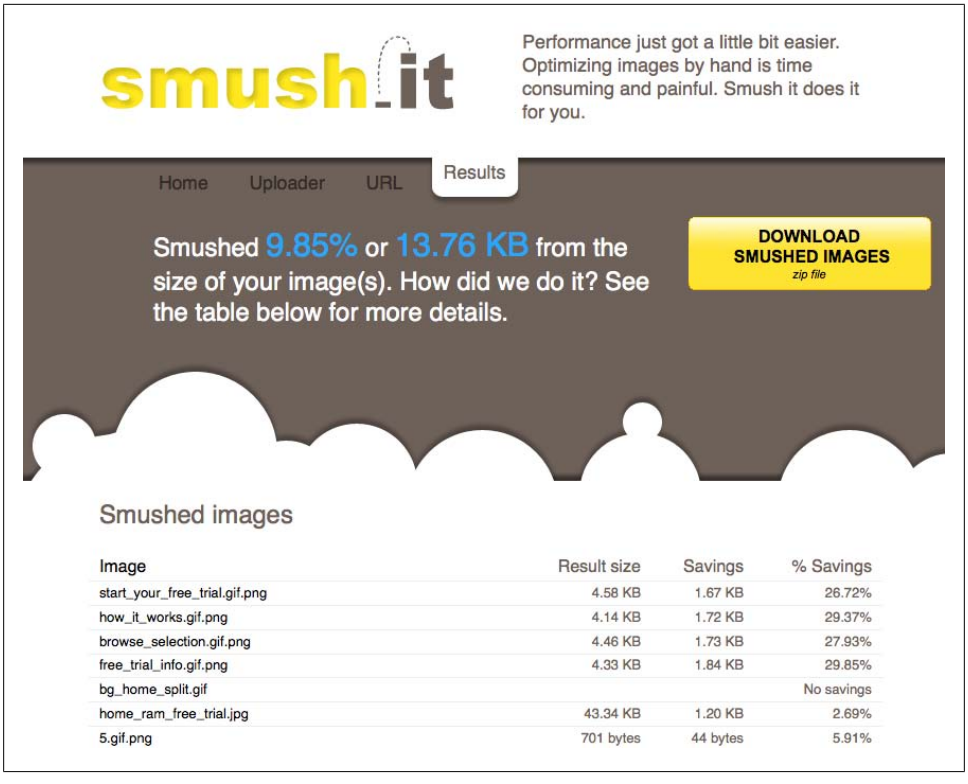


Figure A-6. Smush.it

Cuzillion

Almost every day I wonder about or am asked about a performance edge case. Do external scripts load in parallel if there's an inline script in between them? What if there's an inline script and a stylesheet in between them? Is the behavior the same on Firefox 3.1 and Chrome 2.0?

Instead of writing a new HTML page for each edge case that comes up, I use Cuzillion, shown in Figure A-7. It has a graphical web page "avatar" onto which you can drag-and-drop different types of resources (external scripts, inline scripts, stylesheets, inline style blocks, images, and iframes). Clicking on a resource exposes a variety of configuration settings such as the domain used for loading the resource and how long it takes to respond.

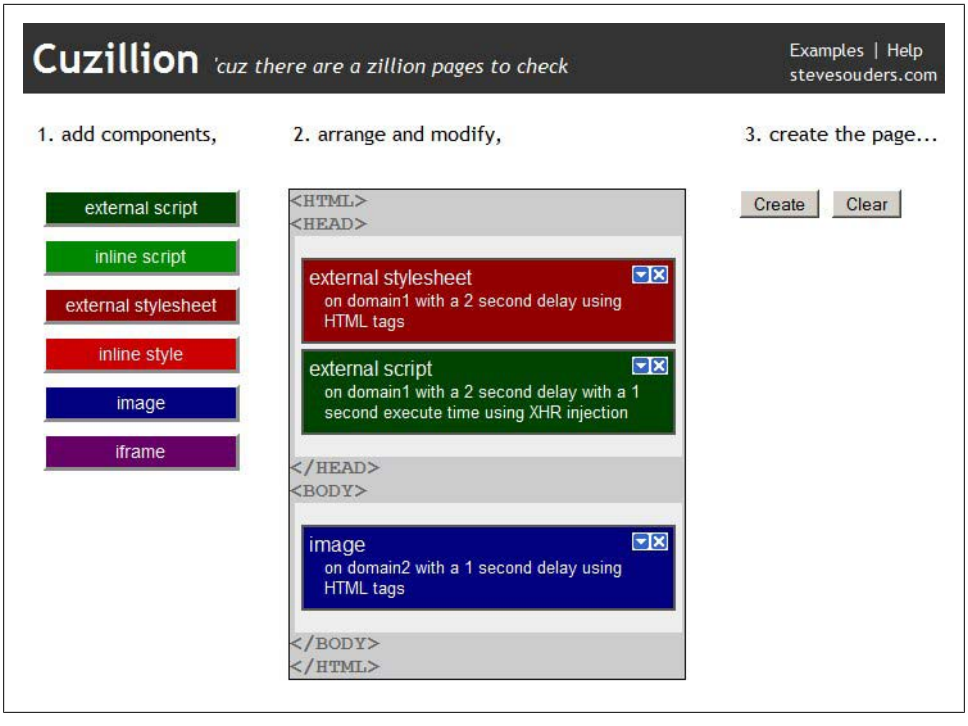


Figure A-7. Cuzillion

I created Cuzillion while I was working on Chapter 4. I needed to test hundreds of test cases. Creating a test framework made this possible in a fraction of the time. The name comes from the tag line: "'cuz there are a zillion pages to check."

UA Profiler

When Google released Chrome, Dion Almaer (coauthor of Chapter 2) asked whether I was going to review it from a performance perspective. Rather than put Chrome through the paces manually, I created a set of HTML pages, each of which contains a specific test: are scripts loaded in parallel, do prefetch links work, and so forth. I then chained those pages together so that the tests would all run automatically.

UA Profiler, shown in Figure A-8, is this set of browser performance tests. In addition to providing a performance test suite for browsers, UA Profiler is also a repository for gathering test results to share with the larger web community. Anyone can point any web client (as long as it supports JavaScript) at UA Profiler and contribute another data point to the results database. By allowing the community to execute the tests, I avoid the cost of running a regression test lab, and also get results under a wider variety of test conditions.

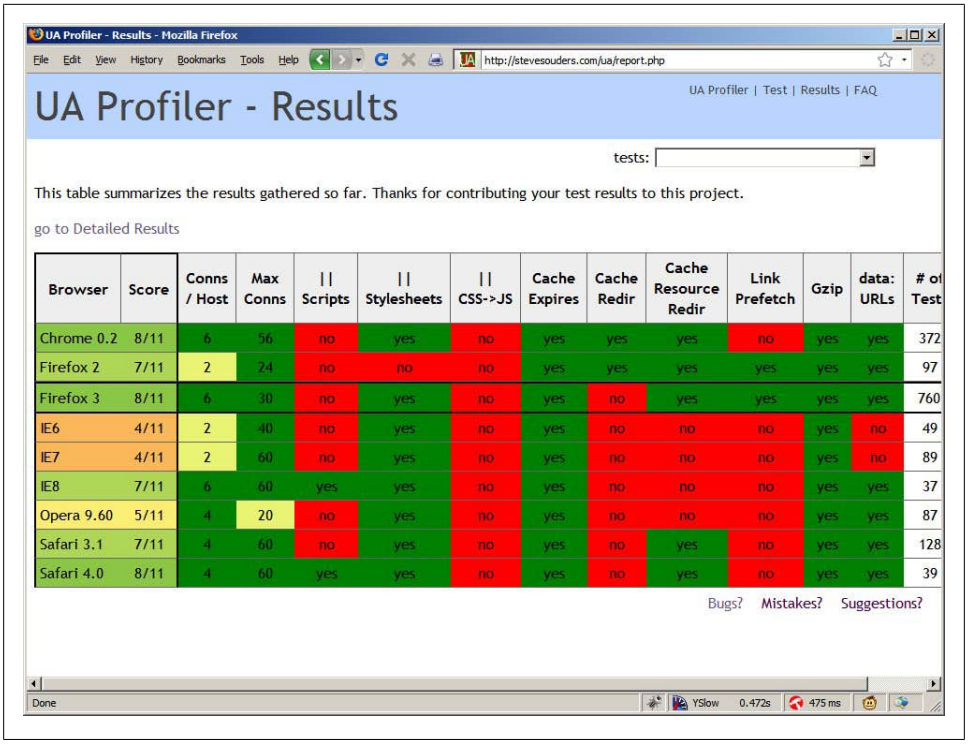


Figure A-8. UA Profiler

For web developers, UA Profiler is useful for confirming how a given browser will perform during a specific optimization. For example, if you're adding future caching headers to a redirect but it still doesn't seem to be cached, you can check UA Profiler to make sure you're using a browser that supports redirect caching.

Index

Symbols

A

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

B

C

D

E

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

About the Author

Steve Souders works at Google on web performance and open source initiatives. His books, *High Performance Web Sites* and *Even Faster Web Sites*, explain his best practices for performance along with the research and real-world results behind them. Steve is the creator of YSlow, the performance analysis extension to Firebug with more than 1 million downloads. He serves as cochair of Velocity, the web performance and operations conference sponsored by O'Reilly, and is cofounder of the Firebug Working Group. Steve taught CS193H: High Performance Web Sites at Stanford, and he frequently speaks at such conferences as OSCON, SXSW, Web 2.0 Expo, and The Ajax Experience.

Steve previously worked at Yahoo! as the Chief Performance Yahoo!, where he blogged about web performance on Yahoo! Developer Network. He was named a Yahoo! Superstar. Steve worked on many of the platforms and products within the company, including running the development team for My Yahoo!. Prior to Yahoo!, Steve worked at several small- to mid-sized startups, including two companies he cofounded, Helix Systems and CoolSync. He also worked at General Magic, WhoWhere?, and Lycos.

In the early 80s, Steve caught the Artificial Intelligence bug and worked at a few companies doing research on Machine Learning, including several publications and conference appearances. He received a BS in systems engineering from the University of Virginia and an MS in management science and engineering from Stanford University.

Steve's interests are varied. He's played basketball with several NBA and WNBA players. He was a member of the Universal Studios Internet Task Force. He participated in setting a Guinness world record. He rebuilt a 90-year-old carriage house. He has a wonderful wife and three daughters.

Colophon

The animal on the cover of *Even Faster Web Sites* is a blackbuck antelope (*Antilope cervicapra*), an endangered species found mainly in India, also known as the Indian antelope. The V-shaped horns of the male blackbuck are ringed with several spiral twists and can be as long as 28 inches. The male's upper body is black or dark brown, and its belly and the rings around its eyes are white. The female is light brown and does not normally have a horn. Blackbucks roam the plains in herds of 15 to 20, feeding on grasses, flowers, and fruits. On the open plain, the blackbuck is one of the fastest animals on earth, able to reach speeds of 45 mph and outrun most predators over long distances.

From the 18th through the early 20th centuries, the blackbuck antelope was the most hunted wild animal in India. In 1932, several species of Indian deer and antelope, including the blackbuck, were introduced to Texas for hunting and breeding. Today, these species live on private hunting ranches and roam the surrounding hill country.

They are so plentiful—having multiplied to 19,000 throughout the state—that many have been shipped to India to repopulate the native habitat.

Now protected in India by the Wildlife Protection Act of 1972, the blackbuck population is steady at 50,000 native animals, plus 43,000 descended from Texas and other populations. Although poaching is still a problem and humans have encroached on its land, its protected status gained attention in 2006 when Indian film star Salman Khan was sentenced to five years in jail for killing two blackbucks. According to Hindu mythology, the blackbuck is considered to be the vehicle of the moon god, Chandrama, and is believed to bestow prosperity wherever it lives.

The cover image is from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.