

/THEORY/IN/PRACTICE

Ajax Design Patterns

Creating Web 2.0 Sites with Programming and Usability Patterns

Michael Mahemoff

O'REILLY®

Ajax Design Patterns

Ajax App Page 59

Produce an Ajax App—a rich application capable of running inside any modern web browser.

Ajax Stub Page 182

Use an Ajax Stub framework that allows browser scripts to directly invoke server-side operations, without having to worry about the details of XMLHttpRequest and HTTP transfer.

Browser-Side Cache Page 290

Retain server results in a Browser-Side Cache.

Browser-Side Templating Page 265

Produce templates with embedded JavaScript and call on a browser-side framework to render them as HTML.

Browser-Side Test Page 554

Create automated tests of browser-side JavaScript components.

Browser-Side XSLT Page 259

Apply XSLT to convert XML Messages into XHTML. Produce templates with embedded JavaScript and call on a browser-side framework to render them as HTML.

Call Tracking Page 210

Track *XMLHttpRequest Calls* as they proceed from browser to server and back again.

Cross-Browser Component Page 283

Create Cross-Browser Components, allowing programmers to reuse them without regard for browser idiosyncracies.

Cross-Domain Proxy Page 243

Create proxying and mediating web services to facilitate communication between the browser and external domains.

DOM Inspection Page 540

Use a DOM Inspection Tool to explore the dynamic DOM state.

Data Grid Page 350

Report on some data in a rich table and support common querying functions.

Debugging Page 537

Diagnose problems with a JavaScript debugger.

Direct Login Page 487

Authenticate the user with an *XMLHttpRequest Call* instead of form-based submission, hashing in the browser for improved security.

Display Morphing Page 69

Morph page elements by altering styles and values in the Document Object Model (DOM), such as text and color properties.

Distributed Events Page 232

Keep objects synchronized with an event mechanism.

Drag-And-Drop Page 390

Provide a drag-and-drop mechanism to let users directly rearrange elements around the page.

Drilldown Page 343

Provide a dynamic Drilldown to let the user locate an item within a hierarchy.

Ajax Design Patterns

Explicit Submission Page 227

Instead of automatically submitting upon each browser event, require the user to explicitly request it; e.g., submit upon a button click.

Fat Client Page 317

Create a rich, browser-based client by performing remote calls only when there is no way to achieve the same effect in the browser.

Guesstimate Page 303

Instead of requesting information from the server, make a reasonable guess.

HTML Message Page 186

Have the server generate HTML snippets to be displayed in the browser.

HTTP Streaming Page 113

Stream server data in the response of a long-lived HTTP connection.

Heartbeat Page 509

Have the browser periodically upload Heartbeat messages to indicate the application is still loaded in the browser and the user is still active.

Highlight Page 468

Highlight elements by rendering them in a consistent, attention-grabbing format.

Host-Proof Hosting Page 493

Host sensitive data in encrypted form, so that clients can only access and manipulate it by providing a pass-phrase that is never transmitted to the server.

IFrame Call Page 108

Use IFrames for browser-server communication.

JSON Message Page 201

Pass messages between server and browser in JavaScript Object Notation (JSON) format.

Lazy Registration Page 474

Accumulate bits of information on the user as he interacts while deferring formal registration until later on.

Live Command-Line Page 376

In command-line interfaces, monitor the command being composed and dynamically modify the interface to support the interaction.

Live Form Page 383

Validate and modify a form throughout the entire interaction, instead of waiting for an explicit submission.

Live Search Page 368

As the user constructs and refines his search query, continuously show all valid results.

Logging Page 534

Provide your JavaScript with log messages.

Malleable Content Page 407

Compose the page of Malleable Content blocks—small chunks of content that can be edited in page.

Microlink Page 416

Provide Microlinks that open up new content on the existing page rather than loading a new page.

Ajax Design Patterns



Michael Mahemoff

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Ajax Design Patterns
by Michael Mahemoff

Copyright © 2006 Michael Mahemoff. All rights reserved. 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 (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Simon St.Laurent

Cover Designer: Mike Kohnke

Production Editor: Mary Brady

Interior Designer: Marcia Friedman

Copyeditors: Mary Brady and Lydia Onofrei

Illustrators: Robert Romano and

Indexer: Julie Hawks

Jessamyn Read

Printing History:

June 2006: First Edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Ajax Design Patterns* and related trade dress are trademarks of O'Reilly Media, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This work is licensed under the Creative Commons License Attribution 2.0. To view a copy of this License, visit <http://creativecommons.org/licenses/by/2.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California 94105-3013, USA.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-10180-5

ISBN-13: 978-0-596-10180-0

[M]

[10/06]

CONTENTS

	PREFACE	ix
Part One	INTRODUCTION	1
1	INTRODUCING AJAX	3
	<i>Ajax and the Usable Web</i>	3
	<i>The Rise of Ajax</i>	5
	<i>Ajaxifying the Web: The Story of Portals</i>	6
	<i>Webifying the Desktop: The Story of Office Applications</i>	9
	<i>Characteristics of Ajax Applications</i>	10
	<i>The Ajax Technologies</i>	14
	<i>Anatomy of a Server Call</i>	15
	<i>Ajax Trends</i>	17
	<i>Conclusions</i>	20
2	A PATTERN-LED TUTORIAL	21
	<i>Ajax Technologies in a Blink</i>	21
	<i>Ajaxifying a Web App: One Pattern at a Time</i>	27
	<i>Projects and Katas</i>	40
	<i>Conclusions</i>	41
3	AJAX DESIGN: PRINCIPLES AND PATTERNS	43
	<i>Desirable Attributes of Ajax Applications</i>	43
	<i>Designing for Ajax</i>	44
	<i>Ajax Patterns Overview</i>	47
	<i>Anatomy of a Pattern</i>	48
	<i>Ajax Patterns Demos</i>	54
	<i>Conclusions</i>	55
Part Two	FOUNDATIONAL TECHNOLOGY PATTERNS	57
4	AJAX APP	59
	<i>Ajax App</i>	59
5	DISPLAY MANIPULATION	69
	<i>Display Morphing</i>	69
	<i>Page Rearrangement</i>	78

6	WEB REMOTING	85
	<i>Web Service</i>	86
	<i>XMLHttpRequest Call</i>	91
	<i>IFrame Call</i>	108
	<i>HTTP Streaming</i>	113
	<i>On-Demand JavaScript</i>	122
7	DYNAMIC BEHAVIOR	133
	<i>User Action</i>	133
	<i>Scheduling</i>	143
8	EXTENDED TECHNOLOGIES	149
	<i>Richer Plugin</i>	149
Part Three	PROGRAMMING PATTERNS	159
9	WEB SERVICES	161
	<i>RESTful Service</i>	162
	<i>RPC Service</i>	177
	<i>Ajax Stub</i>	182
	<i>HTML Message</i>	186
	<i>Plain-Text Message</i>	191
	<i>XML Message</i>	195
	<i>JSON Message</i>	201
10	BROWSER-SERVER DIALOGUE	209
	<i>Call Tracking</i>	210
	<i>Periodic Refresh</i>	215
	<i>Submission Throttling</i>	221
	<i>Explicit Submission</i>	227
	<i>Distributed Events</i>	232
	<i>Cross-Domain Proxy</i>	243
11	DOM POPULATION	253
	<i>XML Data Island</i>	254
	<i>Browser-Side XSLT</i>	259
	<i>Browser-Side Templating</i>	265
12	CODE GENERATION AND REUSE	275
	<i>Server-Side Code Generation</i>	275
	<i>Cross-Browser Component</i>	283

13	PERFORMANCE OPTIMIZATION	289
	<i>Browser-Side Cache</i>	290
	<i>Predictive Fetch</i>	297
	<i>Guesstimate</i>	303
	<i>Multi-Stage Download</i>	310
	<i>Fat Client</i>	317
Part Four	FUNCTIONALITY AND USABILITY PATTERNS	327
14	WIDGETS	329
	<i>Slider</i>	330
	<i>Progress Indicator</i>	336
	<i>Drilldown</i>	343
	<i>Data Grid</i>	350
	<i>Rich Text Editor</i>	355
	<i>Suggestion</i>	361
	<i>Live Search</i>	368
	<i>Live Command-Line</i>	376
	<i>Live Form</i>	383
15	PAGE ARCHITECTURE	389
	<i>Drag-And-Drop</i>	390
	<i>Sprite</i>	394
	<i>Popup</i>	401
	<i>Malleable Content</i>	407
	<i>Microlink</i>	416
	<i>Portlet</i>	421
	<i>Status Area</i>	427
	<i>Update Control</i>	431
	<i>Virtual Workspace</i>	436
16	VISUAL EFFECTS	445
	<i>One-Second Spotlight</i>	446
	<i>One-Second Mutation</i>	455
	<i>One-Second Motion</i>	462
	<i>Highlight</i>	468
17	FUNCTIONALITY	473
	<i>Lazy Registration</i>	474
	<i>Direct Login</i>	487
	<i>Host-Proof Hosting</i>	493
	<i>Timeout</i>	499

	<i>Heartbeat</i>	509
	<i>Unique URLs</i>	515
Part Five	DEVELOPMENT PATTERNS	531
18	DIAGNOSIS	533
	<i>Logging</i>	534
	<i>Debugging</i>	537
	<i>DOM Inspection</i>	540
	<i>Traffic Sniffing</i>	544
19	TESTING	549
	<i>Simulation Service</i>	550
	<i>Browser-Side Test</i>	554
	<i>Service Test</i>	558
	<i>System Test</i>	560
Part Six	APPENDICES	565
A	AJAX FRAMEWORKS AND LIBRARIES	567
B	SETTING UP THE CODE EXAMPLES	601
C	PATTERNS AND PATTERN LANGUAGES	605
D	REFERENCES	607
	INDEX	609

PREFACE

AJAX *DESIGN PATTERNS* IS A REFERENCE FOR DEVELOPERS, DESIGNERS, AND MANAGERS WHO WANT TO know how Ajax is being used in the real world. Ajax is a new label to describe rich, desktop-like Internet applications that run in standard web browsers and do not require any special plugins. Its popularity stems from high-profile Ajax applications like Gmail, Google Maps, and 37signals' Basecamp, and it is becoming popular in the enterprise as well.

The patterns in this book fall into four categories. Foundational Technology patterns overview the raw technologies required for Ajax development, such as the XMLHttpRequest object. Programming patterns expose techniques developers have been discovering to ensure their Ajax applications are maintainable and performant. Functionality and Usability patterns are about the kinds of user-interfaces you'll come across in Ajax applications and the new types of functionality that Ajax makes possible. Finally, Development patterns explain the processes being used to monitor, debug, and test their Ajax applications.

Who Should Read This Book?

You should read this book if you want to:

- Learn what's possible—and what's not—with Ajax, and see how Ajax is being used in the real world.
- Get up to speed with core Ajax technologies such as XMLHttpRequest, the DOM, and JSON.
- Discover the patterns developers are using to produce high-quality Ajax architectures, streamline performance, and improve usability.

Because of the reference-like nature of the patterns, the book is accessible to people from different backgrounds. Developers who want to ramp up on Ajax will be able to begin with the tutorial chapter and the foundational technologies. Those who already have some experience with Ajax will probably gain the most from the Programming and Development patterns. People in less technical roles will be able to take a high-level perspective, looking especially at the Functionality and Usability patterns and the Real-World Examples therein to see what's possible with Ajax.

To follow the technical discussion and code examples, programmers should have some experience with the basics of web development—HTML, form submission, server-side scripting, and so on. Ideally, you should know some JavaScript too, as this book isn't intended to teach you the language, but the writing does take into account that many readers will only have basic familiarity with JavaScript.

Ajax is mostly about what happens in the browser, so the book doesn't assume you know any particular server-side environment. On those occasions where server-side code is involved, the examples are PHP-based and always explained in language-neutral terms.

Who Should Not Read This Book?

If you haven't performed any web development work, you're probably better off looking for an introduction to the basic concepts before jumping into Ajax and these patterns.

Ajax development involves working with a broad range of technologies, including HTML, XML, CSS, JavaScript, and server-side development. This book will help you understand how Ajax relates to each of these and how they are often combined together, but if you're looking to learn any one of these in depth or you're seeking a reference on browser specifics, I'd recommend consulting a specialized text.

About the Examples

All the examples in this book—the tutorial code as well as the Ajax Patterns code examples—can be downloaded from <http://ajaxify.com/run>. They've been developed to be compatible with recent versions of Firefox and Internet Explorer (IE), and have been tested on

Firefox 1.5 and IE 6.0.* Most of the server-side code requires PHP 5 or later and was tested on PHP 5.0.4. The PHP code should run on any standard web server capable of running PHP scripts; Apache 1.3.33 was used for testing. The Wiki Demo requires MySQL and was tested on version 4.1.14-max. See Appendix B for information on installing the demos.

Browser Specifics

True to the aims of Ajax, all of the Ajax Patterns are implementable in any modern, standard browser, and the discussions focus more on issues of general architecture and usability instead of browser specifics. However, portability issues do arise in some cases and are addressed where they are critical to implementing the pattern (as in some of the topics in Part II, *Foundational Technology Patterns*). To maintain a high-level focus, most of these discussions are still limited to IE and Firefox; it's beyond the scope of the patterns to outline the idiosyncracies of each browser and version, for which a specialized reference is advisable.

Organization of This Book

Part I, *Introduction*

The first few chapters are a prelude to the patterns. Chapter 1, *Introducing Ajax*, overviews the Ajax phenomenon and discusses current trends. Chapter 2, *A Pattern-Led Tutorial*, is a quick-start tutorial on Ajax and the Ajax Patterns. The design concepts behind the Ajax Patterns are discussed in Chapter 3, *Ajax Design: Principles and Patterns*, along with an introduction to the patterns themselves.

Part II, *Foundational Technology Patterns*

The Foundational Technology patterns outline the “building blocks” at the heart of any Ajax application. Ajax itself is a pattern, hence the first pattern, Ajax App (Chapter 4), which acts as the root for all the Ajax Patterns. The next few chapters look at the three core technologies involved in creating an Ajax App. Display Manipulation (Chapter 5) patterns are about repainting the user interface, and Web Remoting (Chapter 6) patterns document several alternatives for communicating with the server. The Dynamic Behavior (Chapter 7) patterns are about events—those derived by users and those based on timing. Finally, Extended Technologies (Chapter 8) explain how you can extend an Ajax App with nonstandard technologies.

Part III, *Programming Patterns*

The Programming patterns focus on technical qualities of software, in particular maintainability, robustness, and performance. The first chapter in this part, Chapter 9, provides several alternative strategies for designing Web Services. Also related to Web Remoting, the Browser-Server Dialogue (Chapter 10) patterns look at the flow of information between browser and server; e.g., synchronization techniques. The DOM Population (Chapter 11) patterns cover several strategies for handling DOM population

* It would have been nice to ensure full compatibility for all major browsers (e.g., Safari, Opera), and that's certainly advisable for production systems, but portability was considered orthogonal to the individual demos, each aiming to expose information about a particular Ajax concept.

following a server response. Code Generation and Reuse (Chapter 12) contains a couple of general programming patterns for maintainability and portability. Finally, the Performance Optimization (Chapter 13) patterns are about improving not only speed of updates and communication, but also optimizing the user experience in the face of inevitable delays.

Part IV, *Functionality and Usability Patterns*

The Functionality and Usability patterns are focused on usability of Ajax applications. Chapter 14 introduces a number of widgets that are being woven into many Ajax interfaces. A higher-level perspective is taken by the Page Architecture (Chapter 15) patterns, where the focus is on page layout, content breakdown, and techniques for exposing server-side content. With the popularity of Ajax, the Web is undergoing a rise in visual effects somewhat unprecedented on conventional desktop systems, and the most common effects are described in the Visual Effects (Chapter 16) patterns. The patterns in the final chapter of this part, Functionality (Chapter 17) patterns, are about new kinds of functionality that Ajax makes possible.

Part V, *Development Patterns*

The Development patterns are not “things” in the same sense as the patterns in previous patterns, but “processes” you can use to aid development. Diagnosis Patterns (Chapter 18) helps with troubleshooting and monitoring the health of an Ajax App. Testing Patterns (Chapter 19) helps with testing at various levels and is based on the agile practice of testing as the application evolves.

Part VI, *Appendixes*

There are four appendix sections. Appendix A, the largest appendix, is a listing of Ajax-related frameworks and libraries, many of which are a great aid in implementing some of the patterns described in this book. Appendix B is a set of installation notes for the code examples. Appendix C places the work here in the context of general design pattern theory. Finally, Appendix D lists texts referenced throughout the book.

Conventions Used in This Book

Italics

Used for names of patterns

Constant width

Used for code examples and fragments

Constant width bold

Used for important or significant lines of code

The term “the browser” is shorthand for the entire system at the browser end—not only the browser application (e.g., Firefox), but also the web application running inside it (e.g., Google Maps). When I’m talking about the browser application, I’ll usually call it a “web browser”—it’s fairly obvious from the context anyway. Likewise, “the server” or “the server side” refers to everything on the server—the operating system (e.g., Linux), the web server (e.g., Apache), as well as the application-specific scripts running within.

The terms “ECMAScript” and “JScript” don’t appear—“JavaScript” is meant to cover these terms.

The term “class” is used to describe JavaScript functions that are used in a manner similar to classes in object-oriented systems.

The first figure that appears in each pattern’s section is an illustrated overview of that particular pattern.

Some common acronyms used throughout the book are:

- CSS: Cascading Style Sheets
- DHTML: Dynamic HTML
- DOM: Document Object Model
- HTML: Hypertext Markup Language
- IE: Microsoft Internet Explorer
- XML: eXtensible Markup Language

Conventions in Code Examples

Design diagrams are based on Universal Modelling Language (UML) notation, though less formal in most cases.

In the code examples, a .phtml suffix is used for PHP files that output HTML; all other PHP files (such as business logic modules) end in .php.

As illustrated in Chapter 2, `$( )` is aliased to `document.getElementById( )` to reduce code clutter (inspired by the prototype library—see <http://prototype.conio.net/>).

Some code examples have been reformatted for the sake of clarity.

Safari® Enabled



When you see a Safari® Enabled 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://safari.oreilly.com>.

How to Contact Us

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/ajaxdp/>

There's also web page for this book, containing full draft text for all patterns and links to an increasing collection of audio podcasts about the patterns. In addition, you'll find information on various Ajax resources and errata for this book. The web page is located at:

<http://ajaxpatterns.org>

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>

Acknowledgments

Writing this book online means I received lots of excellent feedback throughout the process and have many people to thank. Which is another way of saying it's highly probable I'll leave someone out! If that's the case, please mail me any corrections.

My editor, Simon St.Laurent, has always been quick to respond to my queries and dealt admirably with a book being written in a somewhat unusual manner. Thanks also to proofreader Mary Brady for leaving no stone unturned, and illustrator Rob Romano for helping to make the content accessible at a glance. I also want to express my appreciation to O'Reilly for taking on the Ajax Patterns and especially for letting me write the book online, and blog and podcast about the content without restriction.

The book grew from a blog post on Ajax Patterns (<http://softwareas.com/ajax-patterns>), and it was the insight of Thomas Baekdal to cover Ajax usability principles that inspired to the initial patterns post (<http://www.baekdal.com/articles/Usability/XMLHttpRequest-guidelines>). I also have Brent Ashley to thank for his discussing the patterns at the initial Ajax summit and in his blog, apparently the trigger for the initial ripple of online interest in the project.

The ideas in this book also owe a great deal to Jesse-James Garrett and his seminal Ajax article, without which you would not be reading these words.

All the book reviewers provided valuable feedback on the online version throughout the writing process as well as offering many detailed comments on the book's draft version. The reviewers were: Cameron Shorter (ADI Limited, Mapbuilder) and Kevin Vinsen (ADI Limited), whose employer, ADI Limited, provided time for them both to conduct the review; Jep Castelein (Backbase), Daniel Czarnecki (Zoltak Consulting), Tony Hill (Thomson Corporation), and Alex Kirk (Blummy.com, WizLite.com). In addition, a big thanks to members of the Software Architecture Group (SAG) at the University of Illinois at Urbana-Champaign for conducting several rich discussions on the patterns and making the audio publicly available (<http://www.softwareas.com/sag-ajax-patterns-review-1>)—the comments certainly made a difference as the editing moved into its final stages. Ralph Johnson, who leads the group, nominated the Ajax Patterns for review, and Brian Foote kept me informed throughout the process. I'm also grateful to everyone who added to the public portions of the AjaxPatterns.org wiki and offered feedback on the online draft via email, blog comments, and in wiki discussions.

As the patterns were discovered from many existing applications, I would like to thank the creators of all the examples featured in the pattern descriptions, many of them true pioneers. Special thanks to those developers who explained aspects of their designs to me: Kevin Arthur (Stream), Richard Cowin and Bill Scott (OpenRico), Donovan Preston (LivePage), Jeremy Ruston (TiddlyWiki), and Tino "Crisp" Zijdel (DHTML Lemmings). The patterns are also based on numerous writings, code dissections, ideas, proofs-of-concept, and direct suggestions. In particular, a number of patterns are only here because of the contributions of the following authors and developers (also mentioned in individual pattern descriptions): Julien Couvreur, James Dam, Abe Fettig (Twisted, Jotspot), Chris Justus, Christopher Kruslicky, Alex Russell (Dojo, Jotspot), Richard Schwartz, Mike Stenhouse, Joel Webber, and Chris Were. In addition, there were several news-focused services that made it a lot easier to locate all this content as it happened (and also helped other people find AjaxPatterns.org!). Among these resources: Niner Niner's AjaxBlog.com, Chris Cornutt's AjaxDeveloper.org (Chris is now with Ajaxian.com), Dion Almaer, Ben Galbraith, and Rob Sanheim at Ajaxian.com (disclaimer: I've since joined the Ajaxian team), Shane Witbeck's AjaxMatters.com, Mike Papageorge's FiftyFourEleven.com, Brian Benzinger's SolutionWatch.com, and Mike Arrington's Techcrunch.com.

Last but not least, heaps of gratitude to my family for their support throughout the writing process.

PART ONE

Introduction

THE FIRST FEW CHAPTERS ARE A PRELUDE TO THE PATTERNS. CHAPTER 1 OVERVIEWS THE AJAX phenomenon and discusses current trends. Chapter 2 is a tutorial on Ajax and the Ajax Patterns. The design concepts behind the Ajax Patterns are discussed in Chapter 3, along with an introduction to the patterns themselves.

CHAPTER ONE

Introducing Ajax

BY NOW, YOU'VE PROBABLY USED AJAX ON SITES LIKE GOOGLE MAPS ([HTTP://MAPS.GOOGLE.COM](http://maps.google.com)) Amazon's A9 search engine (<http://a9.com>), and Flickr (<http://flickr.com>). Despite their different domains, all these web sites make heavy use of Ajax. The technology lets them take a great leap forth towards the richness of standard desktop applications, and in a manner which still respects the established conventions of the Web.

Ajax and the Usable Web

No longer are you forced to wait five seconds a web page to reload every time you click on something. Ajax applications change in real-time. They let you drag boxes around instead of clicking on arrows and typing in numbers. They keep page content fresh instead of forcing you to keep hitting Refresh. They show meaningful animations instead of verbose messages.

At the heart of all this is a growing emphasis on web usability.* Perhaps you've heard the story of the dancing bear—everyone's impressed with it even though its skills quite

* See Thomas Baekdal's "The Usability Revolution is Over—We Won!" (<http://www.baekdal.com/articles/Usability/usability-revolution/>).

frankly wouldn't get the bear into a dance academy;* it makes an impression because it *can* dance and not because of *how well* it well dances. The Web felt like that at first. Suddenly you could read news from the other side of the world, find hints on some obscure game, purchase a rare book. All valuable activities, regardless of how easy or difficult to perform them. Usability? We don't need no stinkin' usability!

Here's what happened: people discovered that any coder and his dog can build the basic functionality (and you don't always need the coder); amid the rush of B2B companies hyping multimillion dollar auction systems, I recall one CTO bragging that his summer students created the same thing for a few thousand bucks. So if companies in a saturated market can't compete on raw functionality, what can they compete on? The things that matter to users. Most of the companies that have survived and prospered—companies like Google, Amazon, and Yahoo!—avoided feature bloat and promoted simple, though not dumbed-down, interfaces. It's no coincidence that each of these companies have been busy incorporating Ajax features to that end. Each of these monster dotcoms not only uses Ajax, but has actually pioneered some of the concepts described in this book. You can throw Microsoft into that list as well.

In addition, a whole new generation of companies has risen on the strength of their simple, intuitive applications. 37signals has a suite of tightly focused applications used daily by a passionate user base. With an innovative photo-sharing interface, Flickr built a community of 1 million photo-sharing users in around 18 months.† Another recent entrant is Odeo, a podcast manager that works as an easy-to-use web application rather than running in the desktop like most of the competition. Like their giant counterparts, these newcomers are big proponents of Ajax and have helped define the concepts behind many of the Ajax Patterns featured in this book.

And then there are all the systems you and I will never see firsthand: the scores of web applications sitting on closed intranets. Like the dotcoms, there remain plenty of dancing bears in this category too. Companies welcomed internal web apps, but mostly for technical reasons, such as easy deployment, monitoring, and remote access—and also because it seemed like the “cool” thing to do. Usability was rarely the driving factor. Just ask a user who's wondering where her keyboard shortcuts have gone in the “new, improved” web interface. I recall one web migration that increased an average customer transaction from 20 seconds to 2 minutes! One reaction has been to throw in the towel and retreat back to the desktop. But many companies have chosen to persist with the Web, accepting the idiosyncracies and using whatever workarounds are necessary to get the benefits of a web

* The dancing bear story is used throughout *The Inmates Are Running the Asylum* (Cooper, 1999), a book decrying the obsession with everything that's “cool” at the expense of true usability.

† The Flickr estimate is based on a projection from a June 2005 article (<http://www.internetnews.com/ec-news/article.php/3512866>) that cites the user base as 775,000 and growing at 30 percent per month, leading to over a million users in June or July. The company launched in February 2004 (<http://www.adaptivepath.com/publications/essays/archives/000519.php>) and was soon acquired by Yahoo!.

platform without the usual problems. It's this spirit that has led to Ajax features evolving in the enterprise, and the popularity of Ajax continues to fuel progress. While many usages will remain hidden, one open example we do have is work performed internally at the Sabre travel company, which led to the open source OpenRico library (<http://looksgoodworkswell.blogspot.com/2005/06/death-to-paging-rico-livegrid-released.html>).

FINALLY, WEB APPS GIVE SOMETHING BACK

An all-too-common phenomenon in the past few years is taking a reasonable desktop application and spending lots of time and money “upgrading” it into a hopelessly unusable web application. In his podcast, software consultant Craig Shoemaker recalls one such experience (<http://polymorphicpodcast.com/shows/architectajax/>):

Ajax is about giving back to the user what they might have had taken away from them when their application got turned into a web app in the first place. I did some consulting for a company that had an old Powerbuilder application, and they had all kind of services for the users. They had things like multi-column dropdown lists and all this user interaction from the user-interface.

What did they do? They created their application as a web app and the users hated it. A lot of the services and the interaction that they were used to were taken away because of the statelessness and the way a web application is architected. With Ajax, we can give some of that back.

In summary, Ajax aims to keep the benefits of the Web, but without sacrificing usability. Most users are now comfortable working inside the browser—meaning that, for some applications, we actually get the best of both worlds: better usability *and* better infrastructure.

The Rise of Ajax

On February 18, 2005, Jesse-James Garrett published an online article “Ajax: A New Approach to Web Applications” (<http://www.adaptivepath.com/publications/essays/archives/000385.php>). The Web was becoming richer and responsive, closing the gap with the desktop. Garrett introduced “Ajax” to label the architecture behind the new generation of rich web apps like Google Maps and Google Suggest. Ajax isn’t a plugin, nor a proprietary technology. It’s an architectural style—a high-level design pattern—composed of many related technologies and ideas.

Ajax technologies and applications were around before Garrett's article labelled them as such, but the article was a tipping point. Just like when the terms, "object-oriented," "agile development," and "postmodernism" began to be used, a converging trend had been given a buzzworthy umbrella term around which a community could form. "Ajax" gave us a label for the systems that were combining several powerful technologies. With this label established, the development community could suddenly share ideas about how the technologies fit together, debate in blogs about different design approaches, build libraries to support these kind of systems, and catalog common patterns.

Strictly speaking, the term is an acronym—"AJAX," for "Asynchronous JavaScript + XML"—although Garrett has noted that other technologies like CSS and DOM are just as important in the Ajax equation. "Ajax" just happens to roll off the tongue a whole lot easier than "Asynchronous JavaScript+CSS+DOM+XMLHttpRequest." Consistent with his original article, Ajax is generally written "Ajax," not "AJAX." That's a mindset, not mere cosmetic detail, because Ajax is a design style and attitude rather than a precise set of technologies; the technologies are whatever happen to let us build the things we want to build. Throughout this book, I refer to Ajax in terms of what it offers users and their organizations. Here's a working definition:

An Ajax application builds on standard web technologies to deliver a rich, responsive, user experience.

If you look at the Ajax poster children like Google Maps and Gmail, it should be apparent how they fit this definition. They rely on nothing more than a standard web browser, be it Internet Explorer (IE), Firefox, Safari, Opera, or several others. The interfaces are rich in that they include input controls and display effects that go well beyond the familiar form-submission paradigm. And they're responsive in that changes happen quickly and incrementally. The definition is there to be applied pragmatically—the last thing you'll hear from me is a big argument about whether or not application X is Ajaxian or not. We'll walk through typical characteristics of Ajax apps later on, but let's now look at some examples of how Ajax is transforming the Web.

Ajaxifying the Web: The Story of Portals

If you Google for "year of the portal", you'll find ample evidence that every year since 1996 has been *the* year of the portal. It's just around the corner, really. The idea has so much promise: the first thing you see when your browser opens up is a personal homepage with "My News" and "My Mail" and lots of other boxes just about "Me." In short, a page created by Me for Me. So why have they never really taken off? One big factor is that most portal interfaces, frankly, are unusable.*

* There are several problems with portals that Ajax can't solve on its own. Most importantly, the fact that portal servers don't know what you're doing in the browser most of the time suggests that some kind of browser plugin is necessary. Nevertheless, Ajax remains the most obvious way to create the Web interface to any such system.

Consider the problems you face in using a legacy-style portal, using Excite.com as an example—many other conventional portals work in a similar manner (Figure 1-1).*

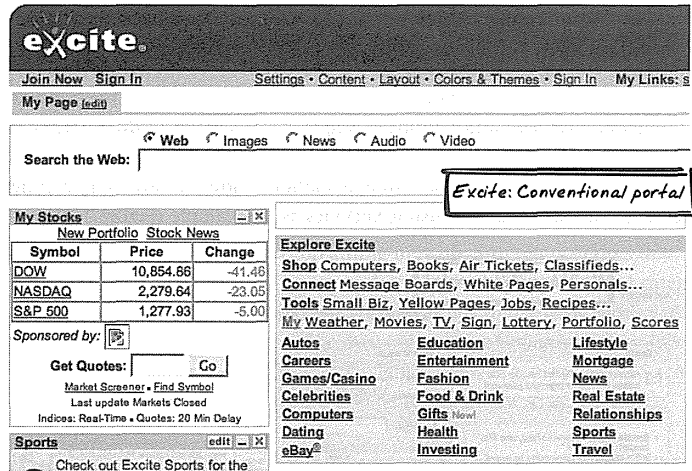


FIGURE 1-1. Excite

1. Customizing the page is the most critical task, but you have to register first; each of the customization controls on the homepage will close the portal and take you to a completely different registration page.
2. Adding new “portlets”—the blocks of content—is rather painful. You have to click on Add/Delete Content, which will whisk you off to a tabbed configuration interface. There, you add and delete portlets by updating a list of current portlets.
3. Customizing an individual portlet—e.g., setting the stocks you’re watching—will close the portal and send you to a special configuration page. You’ve lost the context.
4. Changing layout doesn’t happen directly on the page, but in the configuration area. The layout is managed on a miniature model of the real portal, with titles shown only. (Some portals require repetitive clicking on arrow buttons for layout; fortunately, Excite allows drag-and-drop on the model.)
5. Volatile content such as news and market information is present on the page, but refreshes occur only occasionally; the smallest allowed period is five minutes. Refreshes force the whole page to be reloaded, which is not only distracting, but also makes it difficult for the user to see what, if anything, just changed.

* I’m not picking on Excite, which gained a lot of interest and did some good things with the technology that was available at the time; its inclusion here is testimony to its status as the quintessential example of the first generation of portals.

6. You can't interact with individual portlets—for example, to perform a search. Any time you act on a portlet, such as submitting a form from it, the entire page will update or you'll be sent to a new location.

Portals are so well-suited to Ajaxification that they are probably the most widespread Ajax genre right now; editing the Ajaxian.com blog in late 2005, we reached a point where we were hearing about two or three new Ajax portals a week! Some, like the popular NetVibes (<http://netvibes.com>) and Protopage (<http://protopage.com>) products, are startups. Among the more mature portal producers are none other than Google (<http://www.google.com/ig/>) and Microsoft (<http://live.com>). An explanation follows of how Ajax rectifies each of the problems mentioned above, using NetVibes as an example (Figure 1-2).

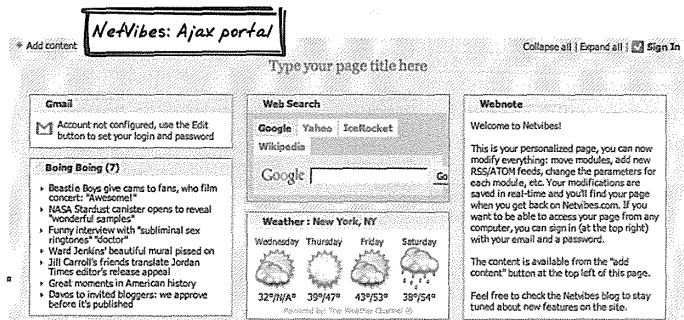


FIGURE 1-2. NetVibes

1. When a new user visits NetVibes, she is free to add and manipulate content, which will stay there for the next time she logs in from the same browser (via cookies). As explained in *Lazy Registration* (Chapter 17), this has always been possible, but Ajax makes the customizations richer and the transition to registering smoother.
2. Clicking on NetVibes' Add Content link doesn't cause a disruptive new page to be shown. It is simply the appearance of a new column (an example of the *Microlink* pattern). There, you can choose new portlet types and watch them appear instantly in the portal. Thanks to the remoting and display manipulation technologies of Ajax, a browser app can talk to the server and update elements without forcing a page refresh.
3. Portlets are customized in-page and without disrupting the other content. Clicking on an Edit link will lead to a small customization form being squeezed into the portlet (an example of *Malleable Content*). There's no page refresh involved, and you can see the effects of editing immediately.
4. Changing layout is as effortless—and fun—as dragging portlets around the page, discussed in *Drag-And-Drop* (Chapter 15).

5. Portlet content is updated directly and without page refresh. Moreover, each portlet is refreshed on its own schedule. In theory, the weather portlet could be updated once a day; the news portlet every five minutes; the stock portlet each second, as explained in *Periodic Refresh* (Chapter 10) and in *Portlet* (Chapter 15). When a portlet updates, an effect like those described in *One-Second Spotlight* (Chapter 16), can be shown for the sake of user feedback.
6. You can have a conversation with an individual portlet, clicking on controls and watching it update. It's as if the portlet is a mini-page; no page refresh occurs and no other content is affected.

The story of portals demonstrates how Ajax can radically improve the usability of a well-established web genre. Indeed, Ajax is breathing new life into many genres that, like portals, had stagnated. Flickr is an Ajax-heavy update of the old photo-sharing category. Gmail (<http://gmail.com>) reinvented webmail, Google Maps (<http://www.googlemaps.com>) reinvented maps, and Google Suggest (<http://www.google.com/webhp?complete=1&hl=en>) opened up new possibilities for search and data entry. Newer genres like RSS readers, wikis, social bookmarking and tagging are also benefiting from Ajax.

Webifying the Desktop: The Story of Office Applications

Attempts to webify office applications are almost as old as the Web itself. It's now part of computer folklore that Netscape's Marc Andreessen exclaimed, in the mid-1990s, that MS-Windows would be reduced to "a poorly debugged set of device drivers running Netscape Navigator," expecting to herald in a new era of desktop-style applications running inside the browser. The benefits of the Web over desktop apps are clear and abundant—e.g., an ability to access data from any web browser in the world, easy upgrading, no tampering of local machines, and better collaboration. However, there are serious problems too, and the most severe is interface. In the past, it's simply been impossible to produce a portable interface that's good enough to justify switching from the desktop.

This is changing quickly though, thanks to Ajax.* A new generation of Ajax office applications (<http://innerphaze.homelinux.com/blog/?p=28>) are emerging as a serious substitute for MS-Word, Excel, and their desktop contemporaries.

One such offering is Writely, a Google acquisition billed as "The Web Word Processor" (Figure 1-3). Writely rightly avoids slavishly reproducing the desktop word-processing experience and instead aims for a feature set and interface that will work on the Web. The result is something that's as much a turbo-charged wiki as a webified word processor. The list that follows describes some of its features.

* As with the portals, Ajax alone is no magic bullet for web-based office apps—there are other issues at stake like interoperability and security of hosted content. That said, Ajax does remove one big barrier by at least making them usable.

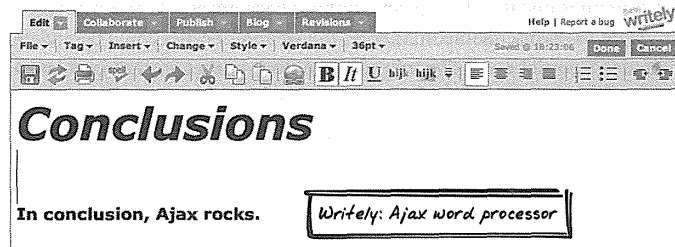


FIGURE 1-3. Writely

- The content under edit is What-You-See-Is-What-You-Get (WYSIWYG). As you edit, you get to see the final content—color, fonts, layout, and all. The idea is covered in *Rich Text Editor* (Chapter 14).
- Writely allows several people to collaborate on the same document at once. Using technology described in the *Periodic Refresh* (Chapter 10) pattern, it's able to keep updating the document and also the status of other authors.
- Documents are easy to manage—you can “tag” (add keywords) and “star” (mark for attention) an element with immediate effect. Again, this is an example of fitting into web conventions. The Web Remoting (Chapter 6) patterns describe making the change persistent without forcing a page refresh.

The story of office applications illustrates there's enough new substance in Ajax to contemplate serious web versions of applications that have been stuck in the desktop forever. In addition to Writely, spreadsheets are being supported by the likes of NumSum (<http://numsum.com>) and presentation managers are being supported by the likes of S5 (<http://www.meyerweb.com/eric/tools/s5/>). Beyond the office, there are Ajax versions of instant messaging clients (see Meebo at <http://meebo.com>), terminal emulators (see Anyterm at <http://anyterm.org/>), and even programming environments (see Why the Lucky Stiff's Try Ruby at <http://tryruby.hobix.com/>). Web applications will always be more limited than their desktop counterparts, but Ajax closes the gap to a point where many of them are suddenly “good enough.” Being “good enough” will trump many users' desktop alternatives, thanks to the intrinsic benefits of working inside the browser.

Characteristics of Ajax Applications

Earlier on, Ajax was defined as a technology that “builds on standard web technologies to deliver a rich, responsive, user experience.” This shouldn't be seen as a binary thing, because it's useful to think of Ajax as a continuous spectrum—an application that happens to include a Flash widget, or one that avoids using any remoting technology can still be considered “partly” Ajaxian; it's useful to do so if you're designing that system as you can leverage the experience of other Ajax developers, which is the kind of experience encapsulated in the Ajax Patterns. And in documenting the Ajax Patterns, it's certainly useful to learn from applications that aren't “pure Ajax.” To that end, the characteristics here are

intended as a general guide, but not hard-and-fast rules, for what constitutes an Ajax application.

Applications, Not Just Web Sites

These days, you'll hear a lot more about "web applications"—or "webapps"—than about "web sites." Driving many modern web projects is the perspective of the browser as a platform and the Web as an operating system. People aren't just buying a book or browsing a manual, but are performing day-to-day work as well as socializing via the browser platform, often working on more critical, complex tasks than in the past. While Ajax can really be applied to anything running inside a browser, it comes into its own with these kinds of systems, where it helps keep users engaged and productive.

Smooth, Continuous Interaction

Traditional web sites make you submit a form, wait a few seconds, watch the page redraw, and then start the whole cycle again. That's because the tiniest server interaction, and even the tiniest display change, requires a call to the server, and then a complete page refresh. It's a frustratingly slow and erratic sequence. Ajax changes the model in a few ways. First, JavaScript running inside the browser can manipulate the display directly—you don't have to send a whole new page from the server in order to hide an element or rearrange the page. Second, server interaction can be handled via JavaScript, so you can upload user commands and download new information without any page refresh. Third, user actions such as mouse-clicking and typing can be handled by JavaScript, so the interaction is a lot richer than just filling in a form and hitting Submit. All of these enhancements make Ajax interaction feel faster and more continuous.

Live

With browser-server interaction no longer a major ritual, it's possible to continuously poll the server for new information. Thus, an Ajax App can be programmed to always show the latest news, details on who else is online, or to send messages to the user. The content is "live."

Supportive

Ajax Apps can monitor user actions and proactively support whatever task the user's working on. Actions as small as a single keystroke can lead to a server call, where the server's knowledge and processing power can be harnessed to produce useful content a subsecond later. For example, a form might change according to the user's input, or an error message might appear as soon as a value is typed.

Visual Effects

Ajax Apps look similar to conventional web apps, but do tend to include a little more animation. Not the kind of flashy animation that's just there for the sake of it, but animation that gets across a message about what's happening and what the user can do next. For example, a deleted icon might slowly shrink and disappear.

WHY WEB APPLICATIONS?

So what's the big attraction of web apps? Why, when the desktop seems to be working just fine, are many developers targeting browsers instead? The following describes some of the reasons why people are moving towards the web platform.

- People are using different computers at home, at work, at school, in cafes, and on their phones. Hosting the data online is the most natural way to take their data and preferences with them.
- A much greater problem than having too many computers is having none at all. Many people around the world have no computer to install desktop software on and store their data. For them, a web application is the only practical way to use a particular application and safely retain all their data.
- Desktop applications suffer from something of a catch-22 situation: a user needs to be convinced an application is useful enough to bother installing, but she often can't make that call until she's installed it. In contrast, most web applications allow a user to jump in straight away and immediately begin using the application, avoiding an installation process altogether.
- Many homes and offices now have broadband, and server hardware is more powerful than ever. The infrastructure makes it possible to deliver the kind of rich, interactive, applications that were envisioned in the 1990s but weren't yet practical. Furthermore, server-side storage is cheap enough for vast amounts of data to be held online—Gmail's initial offering of 1 GB mail storage took the world by surprise, and there are now startups offering to host entire music collections online (e.g., mp3tunes.com).
- The technologies behind Ajax—JavaScript, the DOM, and web remoting—have matured and become more standard. This means web applications can now be made more portable and more usable. In addition, there's a hidden benefit of modern web applications: performance. Compared to the old days of complete page refreshes, smart developers can choose to minimize data transfer with a range of performance optimization techniques. So performance is not only boosted by bandwidth increases, but by the new school of web application architecture.
- For developers, a modern web application is often more productive than a conventional GUI alternative, especially if you want frequent releases on multiple platforms. Developers only have to code a single product for all platforms; they can upgrade the application incrementally rather than in “big bang” style. And on the server, where most of the logic lives, they can use whatever programming language and libraries they care to work with.
- Developing rich applications on the Web used to be considered a kind of rocket science. But it's actually a lot easier now to develop for the Web—arguably easier than many GUI environments. Several factors have improved the development experience: (a) developers are now comfortable with the web architecture and the various libraries and frameworks around; (b) these libraries and frameworks have improved a great deal, especially since the rise of Ajax; (c) browsers are now more consistent and standards-based, and also offer better support for development such as debugging toolkits.

—continued—

- As security concerns have heightened, companies are now quicker to lock down desktops and forbid browser plugins. Web applications hosted on a company's Intranet are often seen as more secure, and support tighter monitoring and access control.
 - Application developers are usually interesting in supporting as wide a user base as possible. Those who just target MS-Windows will not only miss out on other desktop options like Apple and Linux, but also on less conventional platforms like smartphones, home entertainment systems, and game consoles. A web application is often a more flexible way to target these newer platforms.
 - One of the great strengths of the Internet is the ability to communicate and collaborate with remote users. However, doing that relies on common or interoperable software. As web browsers are virtually ubiquitous on Internet-connected computers, web apps are a very attractive option for communication and collaboration.
-

New Widgets

Ajax widgets go beyond the standard HTML controls—text fields, selectors, buttons, and so on. Widgets such as sliders and progress indicators, built on standard HTML elements, are becoming popular. As well, we're seeing conventional widgets enhanced. Instead of a boring old table, you might see a searchable, editable data grid. In place of a textarea element, you might see a rich text editor with "What-You-See-Is-What-You-Get" qualities similar to most word processors.

New Styles of Interaction

It's not only widgets that are getting an upgrade, but styles of interaction. Here, too, Ajax developers have been borrowing from concepts in traditional desktop environments. Drag-and-drop, for example, has been a staple feature of windowing environments for two decades, but somehow didn't make it onto the Web for a long, long time. Now we're seeing it more and more, and it all makes so much sense, as the earlier portal example illustrated. Other styles of interaction are also becoming more popular. Keyboard shortcuts are being used to streamline activity. And some developers are being a bit more adventurous with mouse buttons, experimenting with double-clicking as well as the right and middle buttons. We're yet to see mouse gestures like those available in the Opera browser and other programs, but anything's possible.

Standards-Based

Ajax applications require nothing more than a standard web browser from the past few years, such as Internet Explorer 6 or Firefox 1.0. While the precise browser coverage depends on your objectives, the point is that Ajax makes the most of standard browser features and avoids browser-specific features and plugins where possible. This means more than *standard* technologies; it's also about respecting *standard* user-interface conventions of the Web.

The Ajax Technologies

Here's a quick rundown of the various technologies involved in Ajax. To begin with, there are several that have always been popular on the Web, which are relatively well-understood by the development community. They are still used in Ajax applications, though sometimes in different ways.

HTML/XHTML

As always, HTML provides the structure of a web page. An Ajax App uses an HTML document to show the initial page, and the document is continuously manipulated to change the display and set up new events. Where possible, its XML-compliant variant, XHTML, should be used in order to make manipulation more robust.

CSS

CSS enriches the display and, thanks to stylesheets, helps separate document structure from style details. Fortunately, browsers are now reasonably consistent in their support for CSS2, so the past few years have seen many web sites shift from table-based layout, which was always something of a hack, to the cleaner, more flexible, CSS-based layout. From our perspective, the great thing about all this is that CSS can easily be manipulated with JavaScript. With just one line of code, you can make an object disappear, move it around the page, or alter its appearance.

HTTP, CGI, Form Submission

As with conventional web applications, Ajax communicates via HTTP. The difference is that instead of returning full pages, the server returns concise results that are then processed in the browser script. Form submission—often with CGI-style URLs—is also used, but again is initiated programmatically, meaning that no page refresh need take place.

Server-Side Scripting

The server is still required to perform tasks like data persistence and input validation. However, in some Ajax architectures, it no longer performs any duty of display or application logic, leaving those things for the browser script to handle.

Ajax also adds a mix of newer technologies. I say “newer” with considerable reservation, because every one of them has actually been around for many years. It's just that they're only now becoming well-understood, standard across all major browsers, and combined together to produce a new style of application.

JavaScript

One thing is true about any Ajax App: stuff happens in the browser. JavaScript is the client-side programming language that coordinates browser activity. Prior to Ajax, web developers might have sprinkled a little JavaScript into their pages, but few could claim to have a thorough understanding of the language. That's now changing, with developers making an effort to master JavaScript just as they would strive to learn any given server-side language.

XML and the Document Object Model (DOM)

A DOM object is a hierarchical data structure representing an XML document, and JavaScript can read and manipulate DOM structures. One particularly important XML document is the current web page, since HTML is, loosely speaking, a dialect of XML (for most intents and purposes, browsers will treat it as such, though it's really only when you code in XHTML—an XMLicized form for HTML—that you're truly using XML). Being a kind of XML document, the current web page is exposed as a DOM object in the browser, and by manipulating it, you can affect what's on the page (see *Display Morphing* and *Page Rearrangement* [Chapter 5]). In addition, there will be other DOM objects as well if you choose to talk to the server using *XML Message*.

Event model ("DHTML")

The event model allows JavaScript to respond to events such as mouse clicks. This is the key to interactivity within the browser—a conversation between the user and the web site (see *User Action* and *Scheduling* [Chapter 7]).

Web Remoting

Web Remoting is the ability for JavaScript to talk directly with the server—no page refresh required. XMLHttpRequest dominates most of the headlines, but there are actually several viable remoting technologies (see *XMLHttpRequest Call*, *IFrame Call*, *HTTP Streaming*, and *On-Demand JavaScript* [Chapter 6]).

Anatomy of a Server Call

At a lower level, how does an Ajax interaction look? Here's a typical sequence of events. Let's begin with the application startup sequence (Figure 1-4):

1. User points browser to Ajax App. The user begins interacting with an Ajax application by visiting it in the usual way; e.g., by following a link or selecting a bookmark.
2. Browser loads initial content. The browser fills up with initial content sent out by the Ajax application. This includes the initial HTML to be displayed, the CSS to establish styling, and the JavaScript code to manage further interaction. The HTML is sometimes as raw as a general page structure, in which case the initial content will subsequently be pulled down in a second call. The code will usually set up event handlers to dictate how the system should respond to user actions.

Once the application has loaded, further activity will be triggered by events. Following is the typical sequence for each event (Figure 1-5):

1. User does something. Most events are triggered by user actions such as mouse clicks. As explained in Dynamic Behavior patterns, JavaScript functions can be registered against particular event types on particular page elements; e.g., you can arrange for the `purchase()` function to be called whenever a shopping item (the page element) is double-clicked (the event type). Thus, the user action will typically cause an event handler to be invoked.

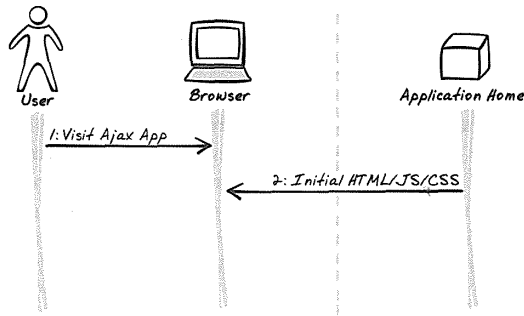


FIGURE 1-4. Typical startup sequence

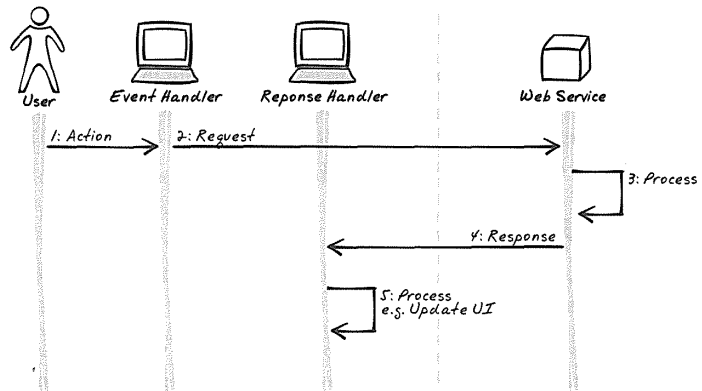


FIGURE 1-5. Typical event-handling sequence

2. Event handler sends request to server. Certain events require server interaction—for example, the user has just entered some information that should be validated by the server; or the user has just requested some new information. XMLHttpRequest—a key Ajax technology—is designed specifically for this purpose. It’s a way for JavaScript code to directly call the server, and in doing so, the code will indicate a callback function that will be notified when the response eventually arrives. Browser-server interaction is described in Web Remoting (Chapter 6) patterns.
3. Server processes request. In the server, a web service (see the *Web Service* [Chapter 6] pattern) receives the request.
4. Server responds. The web service receives the request and outputs a response using standard HTTP techniques that will be familiar to anyone who has worked on traditional web applications.

5. Browser callback function receives response. Back in the browser, the callback function is notified that the call has completed and can read the server response. It's free to use the response however it wants, but the most common activity is changing the display. This relies on manipulating the DOM, a representation of the web page that can be altered with JavaScript. See the Display Manipulation (Chapter 5) patterns for more details.

The sequence gives a quick overview of the technologies—there's a hands-on introduction to each of them in the next chapter, and the first part of the patterns, Foundational Technology Patterns (Part II) covers them in more detail.

Ajax Trends

This book has certainly been a moving target. There were already quite a few Ajax applications available when the "Ajax" term was coined, and the space has since exploded, propelled by the rush of activity in what's come to be known as the "Web 2.0" movement. Each day is bringing fresh ideas to the table, as more and more Ajax applications are released. It's impossible to know where it will all lead, but this section identifies a few future trends and open questions.

Standards and Fragmentation

Better compatibility across browsers has made rich web development much easier. However, we're at a crossroads now, because the same economic boom that's fuelling Ajax application development is also fuelling innovations in the browsers themselves, leading to a serious risk of diverging technologies.

One group pushing for change is the Web Hypertext Application Technology Working Group (WHAT-WG). The key term here is "Application," as the group is pushing for the Web as a true application platform, a goal that resonates loudly with the aims of Ajax. Under current proposals, rich controls and interaction techniques such as drag-and-drop will enjoy native browser support. All this is good for Ajax developers, but will probably come at the price of compatibility. It's not clear which browsers will support which standards, and there are likely to be major discrepancies in the implementation schedules. Moreover, Microsoft is conspicuous by its absence from WHAT-WG, and it's distinctly possible IE will end up with a very different API for all this functionality. As well as standards endorsed by WHAT-WG and W3C, there will inevitably be browser-specific features to consider as well. Microsoft will continue to evolve its Atlas framework, and it's certainly possible that IE will give Atlas functionality—such as local data storage—that's not available to other browsers.

If browsers do go down the mid-1990s path of diverging APIs, developers will have to decide on appropriate strategy. The options will include: targeting a specific browser (unfortunate, but will sometimes be the most pragmatic choice), ignoring browser-specific features, and relying on compatible plugins for behavior not directly supported by a particular browser.

Browser Adoption

There may well be many browser enhancements on the horizon, but will users upgrade to take advantage of those features? Most Internet users haven't heard of Ajax and probably wouldn't care a great deal if you told them about it. They'll need a lot of convincing to upgrade their browser and even greater persuasion in order to switch to a different brand. On the other hand, there's now a large population who grew up with the Internet and are quite comfortable with frequent upgrades and browser switches. Furthermore, browsers—like a lot of desktop software—have become smarter about upgrading themselves, so for users who have installed a recent browser edition that does this, they will always have the latest browser features available.

User Acceptance

Early indications are that users appreciate most Ajax features, but it's still not clear if mainstream users will feel comfortable with features like drag-and-drop and full-blown applications like word processors running inside the browser. Another issue is the handling of URLs and the Back button. The problem is not as bad as some critics have argued, as explained in the *Unique URLs* (Chapter 17) pattern, but it's true that features like page refresh and the Back button don't work exactly the same way as in conventional applications. How will users deal with all these changes, and what tricks will developers discover to help the situation?

The Desktop

Being something of a substitute for the desktop, Ajax's future will be affected by what happens there. Some have predicted the demise of traditional desktop genres as word processors and personal information managers. Certainly, Ajax has closed the gap, and the question is whether the gap will continue to close or widen due to desktop innovation. Microsoft's Live strategy, as well as the rumored Ajax version of the Star Office product from Sun and Google, will be especially interesting to watch, as they essentially involve a hybrid approach: a rich desktop for deep, long-term activity and a lighter Ajax client with all the benefits of a web interface described earlier on.

Flash

Flash has an interesting role in the Ajax world. It's clear that Ajax is stronger from a standards perspective—you can take a regular form and sprinkle some Ajax onto it by way of effects and dynamic validation. Even better, the form can be made to work with or without Ajax technologies available. Flash, however, does many things Ajax can't, such as hardware access, rich graphics, and local file storage. And both technologies are moving targets. Will Adobe steer Flash towards being a direct replacement for Ajax or more of a complementary technology? Or might upcoming browser innovations trump Flash's main selling points?

Rich Graphics

Dynamic vector graphics are now possible. IE supports Vector Markup Language (VML), and Firefox and Safari support SVG. We'll likely see sophisticated APIs evolve and eventually 3-D graphics, opening up even more possibilities. Compatibility issues aside, there are going to be many design issues to consider. In theory, you can generate any display you like, so you could potentially ignore HTML standards altogether and build a set of controls from scratch. That's an extreme example, but developers will face day-to-day decisions between sticking with the standard or opting for a richer custom control, and libraries will likely emerge to provide such controls.

The Two-Way Web

On the Web, it's always the browser that initiates a connection. There's no way for the server to jump into the user's browser and say "Hey, your buddy just logged on!" Well, no standard way anyway; for a long time now, people have been trying every hack they can think of to enable what's known variously as "server push," "streaming," "the two-way web," and, more recently, "Comet" (<http://alex.dojotoolkit.org/?p=545>). These ideas aren't central to Ajax, but there's certainly a growing undercurrent of interest and a great deal of synergy with the core Ajax technologies. A few patterns in this book address the idea (*HTTP Streaming*, *Periodic Refresh* and *Distributed Events*), but it's still difficult to do well, fraught with compatibility issues, and certainly a challenge to scale. Solutions are emerging—servers such as Twisted (<http://twistedmatrix.com/>) that support many long-lived connections, along with browser tricks to make the hacks less intrusive.

Local Storage

As Ajax has become popular, there has become a stronger demand for local storage. Right now, the only standard way to store data inside the browser is with cookies. But people want a way to store much larger data sets—many megabytes—and also to prevent the data ever being transmitted to the server. In addition, local data storage lets users continue to work offline, thus resolving a major argument against rich web applications. Some Ajax libraries, like Dojo, now hook into Flash's local storage capabilities. In addition, there are plans ahead for Firefox to support it, and Microsoft's Atlas framework has also indicated it will do so (it's not clear if the support will be IE-specific). Is local storage really useful when Ajax makes it so easy to synchronize with the server? Will users understand what's local and what's not, and will they be able to manage their data? There will be design issues as well, such as how to handle synchronization after an offline session.

XForms

XForms is a new standard being defined by the W3C, a major upgrade of the standard web form. XForms separate data from view, offer rich widgets for data manipulation, and allow fields to respond to changes in other fields. All of this is already possible with Ajax, but XForms is a more standards-based approach. It's an interesting question as to whether XForms will really take off now that Ajax is so popular, and if it does take off, how will it coexist with Ajax technologies such as web remoting?

JavaScript

Developers have begun taking JavaScript seriously as a result of Ajax, and the language will continue to evolve. In the future, we can expect to see improved support for multithreading, classic interface-implementation separation, and various syntax improvements (see <http://weblogs.mozillazine.org/roadmap/archives/008325.html>). In addition, server-side JavaScript might grow in popularity as it would help with Ajax applications, which often need the same logic and data structures on both sides of the network. For example, validation logic is often run in the browser for better performance, but it must be double-checked in the server for security reasons. In addition to the language evolving, design patterns will also continue to evolve as people push JavaScript to its limits and create more complex scripts.

Development Support

There's already a plethora of Ajax-related libraries and frameworks available (see Appendix A), but we'll likely see a lot more of these develop as Ajax goes mainstream and as the underlying technologies continue to evolve. IDEs and related tools will also evolve—we'll hopefully have the same kind of support for JavaScript that programs like Eclipse and IntelliJ Idea offer today for Java. This includes code refactoring, error checking as you type, and suggestions for code fixes. There are also more specialized, web-related needs, such as traffic visualization and automated web testing. Though tools for these things have been around for a while, they'll need to become a lot more sophisticated to support complex Ajax Apps.

Hardware and Bandwidth Improvements

There have already been surprises with Ajax, such as Google Suggest (<http://www.google.com/webhp?complete=1&hl=en>) showing it's possible to make a round-trip to the server on just about every keystroke. With improving hardware and bandwidth, it will be possible to do even more things previously considered impractical. And more than just bandwidth, latency will also be a critical factor. Might Ajax be practical for real-time multimedia collaboration? Networked games? Interactive data visualization? The answers will depend on the extent of hardware and bandwidth advances, as well as the ability of browser and server software to exploit them.

Conclusions

This chapter has introduced the basic technologies of Ajax, provided overviewed characteristic examples, and given some indication of where it's all going, though there will doubtless be some surprises along the way. Even if the "Ajax" name is fairly new, Ajax has been kicking around in various corners of the Net for some time now, and the rest of the book explains what we've learned so far about working with Ajax. In the next chapter, we'll dive head-first into Ajax with a "pattern-led tutorial"—a hands-on introduction to the basic technologies of Ajax as well as the Ajax Patterns that guide their usage.

A Pattern-Led Tutorial

T HIS CHAPTER INTRODUCES AJAX AND THE AJAX PATTERNS IN A THREE-PART TUTORIAL. THE FIRST section is a whirlwind tour of the key technologies, recommended if you've never worked with Ajax before. The second section shows how the higher-level patterns work to enhance a system; if you've already tinkered with Ajax, you might like to jump straight there. The final section proposes some exercises to try on your own.

The online home for this tutorial is <http://ajaxify.com/tutorial/>, where you'll discover running demos for each example here. To work locally, you'll need a standard PHP installation (version 5 or later) with Apache or a similar web server (no database is required). If you want the completed code, you'll find a downloadable package at the above URL—consult Appendix B for more details on installation. Each section in the tutorial begins with a summary of the corresponding online demo as well as the directory within the code package where you'll find the completed code.

Ajax Technologies in a Blink

This section includes a few exercises to get up to speed with the basic Ajax technologies. Each section is independent from the others, and each corresponds to a group in the Foundational Technologies Patterns (Part II), which are patterns about the raw technologies on which all Ajax applications are based.

Preparing the Demos

Each demo should live in its own directory, so create a tutorial directory under your server's document root, and then create three fresh directories underneath that:

```
cd /apache/docroot (substitute your own document root)
mkdir tutorial
cd tutorial
mkdir display remoting dynamic
```

For Unix systems, ensure permissions are appropriate for your web server (e.g., make the directories globally readable and executable). Each directory will begin with the same “blank slate” HTML file, *index.html*. Open up your favorite editor and save the following file to *display/index.html*, *remoting/index.html*, and *dynamic/index.html*:

```
<html>

<head>
  <title>AjaxPatterns.org - Tutorial</title>
  <script type="text/javascript" src="tutorial.js"></script>
</head>

<body>

  <h1>AjaxPatterns Tutorial</h1>

  <div id="sandbox">
    </div>

</body>
```

Remember to set file permissions according to the web server's requirements. Now point your browser to one of the new files and check that you can see the above content. The URL should be *http://localhost/tutorial/display/* or, failing that, try *http://localhost/tutorial/display/index.html*.

The HTML file loads the JavaScript file *tutorial.js*, which isn't there yet. The next three demos will leave the HTML file alone and illustrate Ajax concepts with code in their respective *tutorial.js* files.

Display Manipulation and the DOM in a Blink

Hello World!

To begin, go to the working directory (tutorial/display/). Note that there's an online demo of the application we're going to build at <http://ajaxify.com/tutorial/display>.

The first thing we want to do with Ajax is update the display—change controls, run animations, and so on. We do that by manipulating the DOM so this tutorial will change the DOM as soon as the page loads. We'll do all our work in the working directory, *tutorial/display*, so once you're in there, save the following as *tutorial.js*:


```

window.onload = function() {
    var greeting = document.createElement("span");
    greeting.style.backgroundColor = "yellow";
    greeting.innerHTML = "Hello World!";
    document.getElementById("sandbox").appendChild(greeting);
}

```

Now point your browser to the *display* demo (<http://localhost/tutorial/display/index.html>) and you should see a “Hello World!” message with yellow background. As explained in *Display Morphing* (Chapter 5), the DOM is a tree structure representing the current page, and the browser will keep the display synchronized with programmatic changes to the DOM. You can, if you like, confirm this by visualizing the DOM after running the script above. There are numerous visualization tools, described in *DOM Inspection* (Chapter 18).

In the above code, `greeting` is a new DOM node that’s appended to the existing `sandbox` DOM node. After creating `greeting`, we set its background color as a CSS style property, and we use `innerHTML` as a simple means of setting its content. The resulting DOM is the same as if the page had been loaded with the following HTML:

```

<div id="sandbox">
  <span style="background-color:yellow;">Hello World!</span>
</div>

```

\$() Convenience function

Now, a quick refactoring. Since we often need the function, `document.getElementById()`, all the demos here will use a convenient alias, so append the following function:

```
function $(id) { return document.getElementById(id); }
```

Now we can make the final line of the earlier code more concise:

```
$("#sandbox").appendChild(greeting);
```

Verify the page works as before. The `$( )` notation* is used throughout this book. In fact, it’s sometimes used repeatedly to refer to the same DOM element, which is a slight performance overhead, since each reference will cause a DOM lookup. In production systems, consider using `$( )` just once to assign a DOM element to a temporary variable, and then reference the temporary variable after that. You’ll gain some performance at the expense of a little code clarity.

Adding a link

We’ll complete this section by dynamically creating a link. Append some code to `onload()`:

```

window.onload = function() {
    var greeting = document.createElement("span");
    greeting.style.backgroundColor = "yellow";

```

* The `$( )` alias for `document.getElementById()` is based on the prototype library (<http://prototype.conio.net/>). Because many popular Ajax libraries build on prototype, it’s becoming something of a standard idiom in Ajax scripts.

```

greeting.innerHTML = "Hello World!";
$("#sandbox").appendChild(greeting);

$("#sandbox").appendChild(document.createElement("hr"));

var link = document.createElement("a");
link.href = "http://ajaxpatterns.org";
link.appendChild(document.createTextNode("More Ajax Stuff ..."));
$("#sandbox").appendChild(link);
}

function $(id) { return document.getElementById(id); }

```

Reload the page and confirm it looks like Figure 2-1.

AjaxPatterns Tutorial

Hello World!

[More Ajax Stuff ...](http://ajaxpatterns.org)

FIGURE 2-1. Tutorial display

We've just added two further child nodes to the sandbox. The first child is an `hr` element for aesthetic purposes. Then comes the link, an anchor tag with appropriate `href` property (just like an HTML `<a href="...">` declaration). To set the link text, we append a new child text node. Contrast this with the earlier example, where we used `innerHTML` to set an element's contents. With `innerHTML`, the browser will parse the HTML content and attempt update the DOM on that basis. It's convenient, but somewhat more error-prone than the kind of direct DOM manipulation performed with the link text.

Web Remoting in a Blink

To begin, go to the working directory (`tutorial/remoting/`). Note that there's an online demo of the application we're going to build at <http://ajaxify.com/tutorial/remoting/>.

We're going to use web remoting to do something very basic: dynamically inject a server-side HTML segment onto the page. First, create the HTML page to be injected: *message.html*:

```

<h2>This is message.html</h2>

and there's <strong>some markup</strong> here<br/>
as well as an <a href="http://ajaxpatterns.org">AjaxPatterns link</a>.

```

Now we have to code *tutorial.js* to retrieve the message and put it in the "sandbox" container. We'll perform remoting with `XMLHttpRequest`, detailed in the *XMLHttpRequest Call* (Chapter 6) pattern. Since its creation is browser-specific, begin *tutorial.js* with a `createXMLHttpRequest()` factory function:

```
function createXMLHttpRequest() {
    try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
    try { return new XMLHttpRequest(); } catch(e) {}
    alert("XMLHttpRequest not supported");
    return null;
}
```

As in the previous demo, create the `$( )` convenience function as well:

```
function $(id) { return document.getElementById(id); }
```

With these library functions in place, the only remaining task is to have `XMLHttpRequest` pull down the content and display it once loaded:

```
window.onload = function() {
    var xhr = createXMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState==4) { // Request is finished
            if (xhr.status==200) {
                $("sandbox").innerHTML = "Retrieved from server ...<hr/>";
                $("sandbox").innerHTML += xhr.responseText;
            } else {
                alert("Message returned, but with error status.");
            }
        }
    }
    xhr.open("GET", "message.html", true);
    xhr.send(null);
}
```

And what you'll see is the contents of *message.html* inside the sandbox, as shown in Figure 2-2.

AjaxPatterns Tutorial

Retrieved from server ...

This is message.html

and there's some markup here
as well as an [AjaxPatterns link](#).

FIGURE 2-2. A message inside the sandbox

The code demonstrates `XMLHttpRequest` usage in a nutshell. Three things happen:

1. `XMLHttpRequest.onreadystatechange` is defined. This is a callback method that's notified several times throughout the call lifecycle, including when the call has completed. For convenience, we've assigned the property to a "closure"—an anonymous method inside `function() { ... }`—to save creating a separate method. The closure's contents are explained below.

2. Next, `XMLHttpRequest.open()` will open up a new connection. The parameters say that a GET request should be sent, and the URL is `message.html`. [See *XMLHttpRequest Call* (Chapter 6) for details on the third argument.]
3. `XMLHttpRequest.send()` will complete the request. The argument is null because the request only has headers, but no body.

How is the `XMLHttpRequest` callback mechanism used here? During a call, `XMLHttpRequest` progresses through several stages, from 0, before the connection's open, to 4, when the response has arrived. Here, we only care about that final state, so `onreadystatechange` ignores any prior states. At that stage, we have a complete response, but is it valid? We have to check that the header has a successful response code, which is 200. If all has gone well, `XMLHttpRequest.responseText` will then hold the content that was retrieved from the server.

So, after the screening the response, `responseText` is a string reflecting precisely the contents of `message.html`: `<h2>This is message.html</h2>...AjaxPatterns link</a>`. It can then be displayed by appending it to the sandbox's `innerHTML`.

A response doesn't have to be HTML. It could also be a piece of data, like "5," "OK," or "1,1,2,3,5." Also, the response handler can use it however it wants to. It doesn't have to place the response directly onto the page, and in fact, it doesn't have to change the display at all—it might just store the value for later on. Furthermore, the target URL could be any location on our server. More often than not, it's a dynamic script rather than a static HTML file like the message file used here. The URL will often contain CGI-style variables as input for the server-side script.

Dynamic Behavior in a Blink

To begin, go to the working directory (/tutorial/dynamic/). Note that there's an online demo of the application we're going to build at <http://ajaxify.com/tutorial/dynamic/>.

This demo shows how to handle events in JavaScript. Begin creating `tutorial.js` with the usual `$( )` definition:

```
function $(id) { return document.getElementById(id); }
```

Once the page has loaded, we'll invite the user to click on the sandbox:

```
window.onload = function() {
    $("#sandbox").innerHTML = "Click Here!<br/>";
};
```

So what we want to do is catch the event of a user clicking on the sandbox. When that happens, we'll append a message to the sandbox:

```
window.onload = function() {

    $("#sandbox").innerHTML = "Click Here!<br/>";
```

```

$("sandbox").onclick = function(ev) {
    ev = ev || window.event;
    $("sandbox").innerHTML =
    Clicked at " + new Date() + ". Event: " + ev + ";

    for (property in ev) {
        var message = "Property " + property + ": " + ev[property] + ";
        $("sandbox").innerHTML += message;
    }

};

```

Clicking on the sandbox triggers the event handler, which outputs the event contents, shown in Figure 2-3.

AjaxPatterns Tutorial

Clicked at Mon Jan 09 2006 18:04:48
GMT+1100 (EST). Event: [object MouseEvent].

Property type: click
Property target: [object HTMLDivElement]
Property currentTarget: [object
HTMLDivElement]
Property eventPhase: 2
Property bubbles: true
Property cancelable: true

FIGURE 2-3. Event contents

We assigned `$("sandbox").onclick` to an anonymous function that will handle the event. That's one common way of registering an event handler. We could just as easily register `$("sandbox").onmouseover` if we want to capture mouse movement instead of clicking activity. Browsers usually generate an event object with information about what just happened, such as the time and mouse location. It's either passed into the callback or set as the global `window.event`, depending on the browser. The first line of the callback ensures `ev` is set to whichever of these is valid, so we can assume for the rest of the callback that `ev` is a proper representation of the event that took place. The handler then shows a detailed report of the event by looping across each property.

Ajaxifying a Web App: One Pattern at a Time

Patterns aren't necessarily about big upfront design; more often, they're used for refactoring and enhancing existing systems. The foundational technologies from the previous section can be seen as techniques for Ajaxifying a conventional web app in addition to being useful for building a new one. In the same way, many of the subsequent patterns can be seen as techniques for improving an existing Ajax App.

Here, we'll look at how a few patterns can be progressively applied to enhance a web app. The conventional app is initially described for comparison, but we really begin building the app at Step 1, where we produce a basic Ajax version. In all, there are four steps. In the packaged code, you'll find the complete code for each step. So if you get lost in the middle of Step 2, you can start Step 3 afresh by taking a clean copy of the completed Step 2 directory. Note that all the application files reside in just the one directory.

Background: Old-School Ajaxagram

There is no working directory (no coding required here). See the location of the completed code within the installation package at /tutorial/ajaxagram/. There's an online demo at <http://ajaxify.com/tutorial/ajaxagram/>.

Starting on familiar ground, the initial version is a conventional web app—no Ajax here. It takes a word and lists all anagrams; i.e., all possible combinations of the letters (Figure 2-4). For simplicity, there's no dictionary comparison to look for real words, and for the sake of performance, the input field is limited to just five characters. We won't actually build this conventional version, but you might want to peruse the code (in *tutorial/ajaxagram/*) for comparison with the next Ajaxified version. There are two source files:

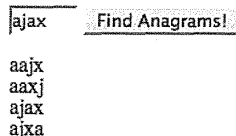
anagram.php

The server-side business logic for finding anagrams, described later in the section "Business logic: the anagram web service."

index.html

The application's view, a query form, and the list of anagrams. The page submits back to itself; the results area always shows all anagrams arising from the previous form submission (if there was one).

Ajaxagram (Conventional Version)



ajax Find Anagrams!

aaajx
aaxj
ajax
aixa

FIGURE 2-4. Conventional Ajaxagram

Step 1: Ajaxagram Done Ajax-Style

Ensure you're in the working directory for this demo (/tutorial/ajaxagram/; this is the same for all steps). Note that you'll find completed code for this step within the installation package under directory /tutorial/ajaxagram/ajaxified/, and an online demo for this step at <http://ajaxify.com/tutorial/ajaxagram/ajaxified>.

The initial Ajax version looks and feels just like the conventional version. The only difference is that there's no page refresh as the query word is passed via web remoting [see the Web Remoting (Chapter 6) patterns], and the results updated using display manipulation [see the Display Manipulation (Chapter 5) patterns]. As an overview, we'll be creating four files inside the one directory:

anagram.php

The server-side business logic for finding anagrams; described in the next section.

anagram.phtml

A server-side web service; accepts a word and outputs a comma-separated list of anagrams.

index.html

The browser-side application's view; a plain HTML file that contains the form and list of anagrams.

ajaxagram.js

The browser-side application logic; waits for a submission, passes the query word to *anagram.phtml*, and alters the display accordingly.

Begin the application by creating a fresh directory; e.g., *tutorial/ajaxagram/* (you'll need to create *tutorial/* if you didn't already do so).

```
mkdir tutorial
cd tutorial
mkdir ajaxagram
```

For Unix systems, ensure permissions are appropriate for your web server (e.g., make the directories globally readable and executable). All the files throughout the four steps described next should go in this directory.

Business logic: the anagram web service

We'll begin the web service by creating a backend module that knows only the business logic related to anagrams, and nothing about HTML. Since we're talking pure business logic, I'll just show you the entire file and skim over the details. Create the following as *anagram.php* (or copy it from the code package):

```
<?

/* Cleans input word and passes into findAnagramsRecursive */
function findAnagrams($word) {
    $sorted = str_split(filterInput($word));
    sort($sorted);
    return findAnagramsRecursive(implode($sorted));
}

/* Convert to lowercase and only include letters */
function filterInput($word) {
    $word = strtolower($word);
```

```

    $word = ereg_replace("[^a-z]", "", $word);
    return $word;
}

/* Assumes $word is sorted */
function findAnagramsRecursive($word) {

    if (strlen($word)==0) { return array(); }
    if (strlen($word)==1) { return array($word); }

    // For each character in the word, extract it and find all anagrams
    // where it's the first character.
    $anagrams = array();
    for ($pos=0; $pos<strlen($word); $pos++) {

        $extractedChar = substr($word,$pos,1);
        // If same as previous, don't process any further as it will only
        // create duplicates (this is the only check against duplicates,
        // since the word is sorted).
        $sameAsPreviousChar =
            ($pos > 0 && $extractedChar==substr($word,$pos-1,1));

        if (!$sameAsPreviousChar) {
            $remaining = removeCharAt($word, $pos);
            $anagramsOfRemaining = findAnagramsRecursive($remaining);
            foreach ($anagramsOfRemaining as $anagramOfRemaining) {
                array_push($anagrams, $extractedChar . $anagramOfRemaining);
            }
        }
    }

    return $anagrams;
}

/* Return word without character at pos */
function removeCharAt($word, $pos) {
    return substr($word, 0, $pos) . substr($word, $pos+1);
}

?>

```

There's also a packaged test case, `AnagramTest.php`; it's good practice to test backend business logic in isolation. If you want to run the test, install PHPUnit2 (http://www.phpunit.de/pocket_guide/3.0/en/index.html) first and see its documentation for details on running tests.

With the business logic now in the server, we need a thin wrapper to expose it as a clean *Web Service*. Placing it in a separate file helps separate business logic from concerns about how it will be invoked from the browser. Thus, create the web service `anagrams.phtml` as follows:

```

<?
    require_once("anagram.php");

```



```

$word = array_key_exists('word', $_GET) ? $_GET['word'] : "";
$anagramsArray = findAnagrams($word);
print implode(", ", $anagramsArray);
?>

```

You can test the web service: point your browser at <http://localhost/tutorial/ajaxagram/anagrams.phtml?word=ajax>, and you should see a comma-separated list of “ajax” anagrams. Notice that this is a raw response intended for easy script processing and not the sort of thing you’d show a user. Our script will need to parse it in order to present the results to users.

Presentation: initial HTML

The initial HTML file contains the basic display detail—a form and a list of anagrams—and also pulls in the required JavaScript. Create *index.html* as follows:

```

<html>

<head>
  <title>AjaxPatterns.org - Ajaxagram (Basic Ajax Version)</title>
  <script type="text/javascript" src="ajaxagram.js"></script>
</head>

<body>

  <h1>Ajaxagram (Basic Ajax Version)</h1>

  <div>
    <input type="text" id="word" size="5" maxlength="5" />
    <button id="findAnagrams">Find Anagrams!</button>
  </div>

  <div id="results">
  </div>

</body>

```

Point your browser to the HTML; you should see the heading and the form, but probably not the results area as it’s initially empty.

There are a few things to note about the HTML, and you might like to contrast it with *index.phtml* packaged in the conventional version. First, the “form” is not a true HTML `<form>` element because we don’t need the browser to submit it to the server; our script will issue the query manually. Second, as there’s no HTML form here, a button element is used instead of a submit control. Third, the results div is empty. It will be populated each time our script receives a list of anagrams from the server. Finally, note the absence of any JavaScript code here; all the application logic is isolated in a separate file, *ajaxagram.js*.

Application logic: JavaScript

The final task is coding the application logic that acts as the glue between the server-side web service and the browser-side web page. Create *ajaxagram.js* with the following generic

methods, explained earlier in the “In A Blink” tutorials. Keep these at the end of the file as we build the rest of the JavaScript.

```
// -- Generic functions -----

function createXMLHttpRequest() {
    try { return new XMLHttpRequest(); } catch (e) {}
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
    try { return new XMLHttpRequest(); } catch(e) {}
    alert("XMLHttpRequest not supported");
    return null;
}

function $(id) {
    return document.getElementById(id);
}
```

The JavaScript needs to handle the query lifecycle, which has three stages:

1. The user clicks the findAnagrams button.
2. XMLHttpRequest submits word input to the anagrams.phtml web service.
3. When the anagram list arrives from server, the anagrams block is updated.

We’ll now create three JavaScript functions, with each function roughly corresponding to each of these stages. The first of these ensures the findAnagrams button will trigger the query:

```
window.onload = function() {
    $("findAnagrams").onclick = function() {
        submitWord();
    }
}
```

If you like, you can test the application now. Upon submitting, you should see an error message because submitWord() is not there yet. We’ll create submitWord() now, using essentially the same boilerplate code as in the earlier “Web Remoting in a Blink” tutorial.

```
function submitWord() {
    var xhr = createXMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState==4) { // Request is finished
            if (xhr.status==200) {
                var anagramsCSV = xhr.responseText;
                updateAnagramsDisplay(anagramsCSV);
            } else {
                alert("An error occurred.");
            }
        }
    }
    var word = $("word").value;
    xhr.open("GET", "anagrams.phtml?word="+word, true);
    xhr.send(null);
}
```

After adding `submitWord()`, the browser submits the query word to `anagrams.phtml`. You can check it by clicking on the Find Anagrams! button and verifying that your web server logs the call to `anagrams.phtml`. You might also monitor the browser-server traffic using a tool mentioned in *Traffic Sniffing* (Chapter 18). Once the script has returned successfully, `updateAnagramsDisplay()` is called.

`updateAnagramsDisplay()` receives the comma-separated list of anagrams, splits it to form an array, and outputs each anagram on a separate line:

```
function updateAnagramsDisplay(anagramsCSV) {
    $("results").innerHTML = "";
    var anagrams = anagramsCSV.split(",");
    for (var i=0; i<anagrams.length; i++) {
        $("results").innerHTML += anagrams[i] + "<br/>";
    }
}
```

Ajaxagram should now work as described earlier. Congratulations, you've built a working Ajax application! You built it using several of the patterns described in the first part of the Ajax Patterns language—the patterns of foundational technologies. Those patterns are enough to build an application, but the patterns of the latter sections consider real-world issues like usability and maintainability. Those are illustrated in the next few sections.

Incidentally, you might notice a small bug. If you search too quickly, you'll see the results for different searches blending together. This is a consequence of the asynchronous nature of Ajax, with results being processed simultaneously in the browser. We'll eventually fix the problem in Step 3, where we introduce one of several possible techniques to prevent issues like this.

Step 2: Enhancing Functionality and Usability

Ensure you're in the working directory for this demo (/tutorial/ajaxagram/; this is the same for all steps). Note that you'll find completed code for this step within the installation package under directory /tutorial/ajaxagram/ajaxified/richer, and an online demo for this step at <http://ajaxify.com/tutorial/ajaxagram/ajaxified/richer/>.

So far, Ajaxagram is impressive insofar as you can now tell your tech friends you've coded an Ajax application. Still, your boss/client/shareholder is going to have a hard time telling it apart from the conventional version. We're going to have to jazz the user interface up if we want to make a true impression. (And hopefully improve usability in the process.) We'll refactor according to Functionality and Usability Patterns (Part IV). By the way, we're only "refactoring by patterns" for demonstration purposes; in real life, it's a lot wiser to let the requirements drive the changes you make and let the patterns follow from there.

Note that the tutorial code package includes the completed files for each step. If you had any problems with the initial version, simply start from here using the files in `/tutorial/ajaxagram/ajaxified/`.

Live Search

We'll begin the UI face-lift by making the input mechanism become a *Live Search* (Chapter 14). No more pressing the Find Anagrams! button because the results will update on each keystroke. Incidentally, Live Search needs to be applied with caution in the real world, because it can sometimes be more of an annoyance than an aid.

Where we previously submitted upon button click, we now submit upon each keypress:

```
window.onload = function() {  
    $("word").keyup = function() {  
        submitWord();  
    }  
}
```

And that's that! Reload the page and we now have a working *Live Search*. Frankly, there's more work required to make it production-ready, but it's pretty neat that our basic Ajax code could be modified so easily to do it. Complete the Live Search by scrapping the Find Anagrams! button, which is now obsolete. This will leave the input area as follows:

```
<div>  
    <input type="text" id="word" size="5" maxlength="5" />  
</div>
```

Progress Indicator

The input field is limited to five characters because long queries take a very long time. But even with five characters, the delay will be noticeable, especially in a real-world context, so we'll ease the pain with a *Progress Indicator* (Chapter 14). This is an animated GIF that's always present on the page, with its visibility toggled on and off, depending on whether we're waiting for a response.

```
<div>  
    <input type="text" id="word" size="5" maxlength="5" />  
</div>  
  

```

You can leave the image URL pointing to my ajaxify.com server—you're welcome to do so and it will work fine—or alternatively, download the image (or copy it from the code package *tutorial/resources/*) to the current directory and reference it locally (using `src="progress.gif"`).

Next, we're going to introduce a CSS file with the styles for waiting and notWaiting modes. Just as we've separated JavaScript from HTML, we'll also separate CSS. To that end, add the following line near the top of `index.html`:

```
<head>  
    <title>AjaxPatterns.org - Ajaxagram (Basic Ajax Version)</title>  
    <script type="text/javascript" src="ajaxagram.js"></script>  
    <link rel="stylesheet" type="text/css" href="ajaxagram.css"/>  
</head>
```

Now create the new CSS file, *ajaxagram.css*, in the same directory.

```
.waiting {
  visibility: visible;
}

.notWaiting {
  visibility: hidden;
}
```

Now we can just flip the image's CSS class between *waiting* and *notWaiting*, according to the request state. The image enters waiting mode when a query is submitted, in the *submitWord()* method, and enters *notWaiting* when the query returns.

```
function submitWord() {
  // Show progress indicator and clear existing results
  $("#progress").className = "waiting";
  $("#results").innerHTML = "";
  ...
}

function updateAnagramsDisplay(anagramsCSV) {
  $("#progress").className = "notWaiting";
  var anagrams = anagramsCSV.split(",");
  ...
}
```

At this stage, you should see the *Progress Indicator* while waiting (Figure 2-5). If the wait is too short to see it, buy yourself a slower PC or alternatively, instrument *anagrams.phtml* with a delay; e.g., `sleep(1);`. That's a basic illustration of the *Progress Indicator* pattern. For a harder problem, increase the width of the input field to allow longer anagrams, and then change the *Progress Indicator* to show percent complete. (Hint: use a *Guesstimate* [Chapter 13] or create a monitoring channel.)

Ajaxagram (Richer UI)



FIGURE 2-5. Ajaxagram with Progress Indicator

One-Second Spotlight (Yellow Fade Technique)

Along with *Progress Indicators*, one of Ajax's great visual landmarks is the *Yellow Fade Technique*, or the more general *One-Second Spotlight* (Chapter 16) pattern. Here, we're going to fade results in each time they update. The div will suddenly become yellow and gradually

settle back to white over the space of a second. It will happen frequently enough to be considered an abuse of the technique, but hey, it's nice eye candy and good enough for demo purposes.

The fade animation here is super-simplistic. In 10 steps, each 100 milliseconds apart, it pumps up the RGB value from (100%, 100%, 0%) to (100%, 100%, 100%). That is, from yellow to white. Thus, red and green components of the div color are fixed at 100 percent, while blue increases by 10 percent each iteration:

```
// Caller-friendly front-end to fadeLoop
function fade() {
    fadeLoop(0);
}

function fadeLoop(nextBluePercent) {
    $("results").style.backgroundColor = "rgb(100%, 100%, "+nextBluePercent+"%";
    nextBluePercent += 10;
    if (nextBluePercent<=100) {
        setTimeout("fadeLoop('"+nextBluePercent+"'", 100);
    }
}
```

Invoke `fade()` just prior to outputting the fresh anagram list:

```
function updateAnagramsDisplay(anagramsCSV) {
    $("progress").className = "notWaiting";
    fade(0);
    ...
}
```

Now click on Find Anagrams! and enjoy the hypnotic Ajax fade effect (Figure 2-6).

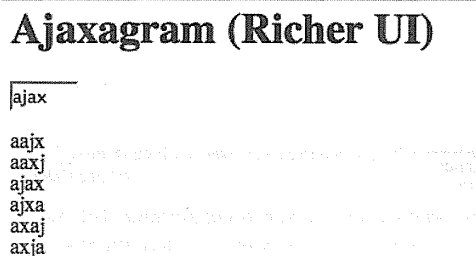


FIGURE 2-6. Yellow Fade Effect (One-Second Spotlight) in Ajaxagram

Step 3: Refactoring for Performance

Ensure you're in the working directory for this demo (`/tutorial/ajaxagram/`; this is the same for all the steps). Note that you'll find completed code for this step within the installation package under directory `/tutorial/ajaxagram/ajaxified/richer/performant/`, and an online demo for this step at <http://ajaxify.com/tutorial/ajaxagram/ajaxified/richer/performant/>.

The previous section shows how the Ajax Patterns go beyond foundational technologies to look at usability and functionality issues. That alone would be useful, but we still need to address the nonfunctional aspects of a rich web application, like performance, data integrity, testability, and programmer productivity. That's the domain of Programming Patterns (Part III), as well as Development Patterns (Part V). Here we'll treat the application with *Submission Throttling*, a programming pattern.

The problem we're dealing with arises from the *Live Search* capability. Responding to each keystroke is nice, but unfortunately impractical in most real-world settings. We want a way to limit—throttle—the submissions. The solution is very simple: submit only after an idle period of one second. Modify *ajaxagram.js* as follows:

```
var submitTimer;

window.onload = function() {
  $("word").onkeyup = function() {
    clearTimeout(submitTimer); // Clears any previously scheduled submission
  }
}
```

Reload the application and you should see the delay between typing the word and seeing the results appear. We've now capped the number of browser-server calls to one per second, which is one style of *Submission Throttling*. It works using timers; as explained in the *Scheduling* (Chapter 7) pattern, JavaScript lets us set a one-off event in the future and cancel it too. The above code ensures that each keystroke schedules a call in one second and cancels any pending call.

Step 4: Refactoring to an External Library

Go to the working directory (/tutorial/ajaxagram/; this is the same for all the steps). Note that you'll find completed code within the installation package under directory /tutorial/ajaxagram/ajaxified/richer/performant/library/, and on online demo for this step at <http://ajaxify.com/tutorial/ajaxagram/ajaxified/richer/performant/library/>.

The final refactoring isn't actually related to a particular pattern, but is a reminder of the importance of libraries and frameworks throughout the patterns. While most patterns cover low-level JavaScript concepts that are worth knowing, that doesn't mean you have to hand-code everything yourself. There's an ever-growing list of production-ready components available, as documented in Appendix A. The demo here shows how we can achieve the same functionality by reusing a library.

A prime candidate is the remoting performed by Ajaxagram. There's very little reason to directly use XMLHttpRequest nowadays, so we'll refactor to one of the many libraries that wraps it in a clean API. We'll use ajaxCaller (<http://www.ajaxify.com/run/testAjaxCaller/>), a small component developed specifically for the demos in this book.

Add a new <script> tag to *index.html* to include the library:

```
<head>
<title>AjaxPatterns.org - Ajaxagram ()</title>
```

```

<script type="text/javascript" src="ajaxagram.js"></script>
<script type="text/javascript" src="http://ajaxify.com/tutorial/resources/
  ajaxCaller.js"></script>
<link rel="stylesheet" type="text/css" href="ajaxagram.css" />
</head>

```

The above line will pull the script down from my ajaxify.com server; as with the progress indicator earlier, you're welcome to leave it like that.* Alternatively, you can download the script (or copy it from the code package *tutorial/resources/*) to the current directory and change the URL accordingly (use `src="ajaxCaller.js"`).

`submitWord()` now becomes almost trivial:

```

function submitWord() {

    // Show progress indicator and clear existing results
    $("progress").className = "waiting";
    $("results").innerHTML = "";

    var word = $("word").value;
    ajaxCaller.getPlainText("anagrams.phtml?word="+word,updateAnagramsDisplay);
}

```

In addition, you can remove the generic `createXMLHttpRequest()` function. That's a lot of code we no longer need thanks to the external library.

Refactoring Exercises

The final Ajaxagram makes a simple platform on which to play around with further patterns. Here are a few exercises you can try out. Let me know how it goes (contact me at michael@mahemoff.com). I'd be pleased to link to any examples from the tutorial's homepage.

Foundational technology patterns

On-Demand JavaScript (Chapter 6)

Instead of downloading all of *ajaxagram.js* at the start, simply download the bootstrapping code and have it fetch the rest of the script on demand.

Programming patterns

Web Services (Chapter 9) patterns

Experiment with the response emerging from the *anagrams.phtml* web service. Try returning an *HTML Message* (Chapter 9) or an *XML Message* (Chapter 9) instead.

Call Tracking (Chapter 10)

As the first step mentioned, the anagrams list sometimes contains results for several queries at once. The refactoring to the *Submission Throttling* (Chapter 10) pattern basically fixed this by enforcing a gap between submissions. However, you don't always

* As explained in the *On-Demand JavaScript* (Chapter 6) pattern, it's perfectly feasible to reference an external JavaScript file. However, beware that there's always some risk involved as you're executing code from a server you have no control over.

want to use Submission Throttling; even if you do, there's no guarantee it will solve this problem. Introduce Call Tracking for a more direct attack.

Submission Throttling (Chapter 10)

We applied one kind of Submission Throttling, but try for the kind discussed in the Code Example section of Chapter 10.

Cross-Domain Proxy (Chapter 10)

Instead of calling your own server for the anagrams, relay the call to my server at ajaxify.com/tutorial/ajaxagram/ajaxified/anagrams.phtml.

Browser-Side XSLT (Chapter 11)

Transmit the anagrams in XML format and convert them with XSLT for display.

Browser-Side Templating (Chapter 11)

Show the results using a template framework.

Browser-Side Cache (Chapter 13)

Cache previous anagram lists.

Predictive Fetch (Chapter 13)

For the basic version, which uses a button for submission, make a pre-emptive call in anticipation of a button press.

Fat Client (Chapter 13)

Forget the server altogether and compute the anagrams in the browser.

Functionality and Usability patterns

Data Grid (Chapter 14)

Show the list of anagrams in a Data Grid, allowing for search, filtering, and so on.

Suggestion (Chapter 14)

Suggest words from a dictionary as the user types.

On-Demand JavaScript (Chapter 6)

Instead of downloading all of *ajaxagram.js* at the start, simply download the bootstrapping code and have it fetch the rest of the script on demand.

Popup (Chapter 15)

Produce the results in a Popup.

Portlet (Chapter 15)

Encapsulate the search as a Portlet and slap it on a web site you maintain.

Virtual Workspace (Chapter 15)

The demo produces many, many results for long queries, which is why the input field is limited to five letters. Allow more letters and show the results as a Virtual Workspace. You'll probably want to make the algorithm itself more flexible.

Unique URLs (Chapter 17)

Make sure each query corresponds to a unique URL.

Development patterns

Diagnosis (Chapter 18) patterns

Apply the various tools mentioned in this chapter.

Service Test (Chapter 19)

Create a standalone HTTP client to test *anagrams.phtml*. Note that *AnagramTest* already tests the backend logic, so the most important thing is simply to check the service interface works.

System Test (Chapter 19)

Apply the various tools mentioned in System Test.

Projects and Katas

Now it's over to you. Here are some suggestions for projects you can work on. You might like to approach these as "Katas" (<http://blogs.pragprog.com/cgi-bin/pragdave.cgi/Practices/Kata/>), short exercises that you can keep returning to. Alternatively, add all the bells and whistles and make them full-blown projects. Check <http://ajaxpatterns.org/Katas> for the latest list, and please add links to any of your online efforts.

Two-Person chat

Allow two people to chat in real-time via the browser.

Filesystem navigator

Navigate a server-side filesystem. Bonus points for sanely displaying file contents.

Search portlet

Build a portlet to access your favorite web service, using a *Cross-Domain Proxy* (Chapter 10) pattern or an *On-Demand JavaScript* (Chapter 6) pattern if the web service happens to offer a compatible service (such as Yahoo's JSON API). Results should be displayed inside the portlet, without any page refresh.

Drag-And-Drop cart

Create a shopping cart that allows items to be dragged in and out and uploads details to the server. Note that several libraries support drag-and-drop.

Am I Ajax Or Not

Create a clone of *AmIHotOrNot.com*. Show two random images (or even just random numbers), let the user click on their favorite, and ensure the server records their choice. For bonus points, show scores in the browser; e.g., offer a popup showing the number of "wins" for the two items being displayed.

Whack-A-Mole game

"Moles" appear at random places on the page and the user must quickly eradicate them by clicking on them. Provide a timer and report the user's score.

Image slideshow

Present a sequence of images using Ajax for navigation. You might also include some visual effects.

Web Site Tuning bookmarklet

Build a bookmarklet that allows the user to tweak the page's style settings; pop up a dialog with fields such as background color and font size. Bookmarklets (<http://en.wikipedia.org/wiki/Bookmarklet>) aren't regular Ajax applications, but they are still a great way to play with Ajax technologies.

Conclusions

The first part of this chapter showed you how to create a basic Ajax application. After performing that section, you should be comfortable about the basics of Display Manipulation (Chapter 5), Web Remoting (Chapter 6), and Dynamic Behavior (Chapter 7). The second part of this chapter introduced a number of the Ajax Patterns—the point is not to understand those patterns in detail, but to get a feel for the sort of thing the patterns do. That is, to see how they build on top of the basic technologies to improve usability, maintainability, and so on. The final part of this chapter suggested several exercises—I recommend you try one or two if you're still coming to grips with Ajax.

It should be clear by now that although Ajax is a great leap forward for the Web, it's no magic bullet. Developers need to tread carefully in order to ensure their Ajax apps are easy to maintain and easy to use. The next chapter summarizes several key principles for Ajax system design and introduces the Ajax Patterns, which offer practical advice on applying those design principles.

Ajax Design: Principles and Patterns

AJAX DOES A LOT FOR WEB USABILITY, HAS ALREADY DELIVERED SOME STUNNING APPLICATIONS, AND IS clearly the “in” thing at this time. But it’s no magic bullet. Careful design is always required, and it must be tailored to the technology at hand. By monitoring the state of Ajax applications, we continue to learn about what works and what doesn’t, and about how developers are succeeding in their design trade-offs. This chapter explains these lessons at a high level and introduces the patterns, which discuss them in depth.

Desirable Attributes of Ajax Applications

Ajax is about improving user experience and delivering value to the organizations that own and use web applications. Here, we’ll look at the key attributes of an ideal Ajax application. Reality dictates that you’ll never get the best of all worlds, so you’ll have to make trade-offs based on how important you consider each attribute. The Ajax Patterns are intended to help you deal with these trade-offs.

Usability

Ajax applications should be as intuitive, productive, and fun to use as possible.

Developer productivity

Development should be as efficient as possible, with a clean, maintainable code base.

Efficiency

Ajax applications should consume minimal bandwidth and server resources.

Reliability

Ajax applications should provide accurate information and preserve the integrity of data.

Privacy

While user-generated data can and should be used to improve the user experience, users' privacy should also be respected, and users should be aware of when and how their data is used.

Accessibility

Ajax applications should work for users with particular disabilities and of different ages and cultural backgrounds.

Compatibility

As an extension to accessibility, Ajax applications should work on a wide range of browser applications, hardware devices, and operating systems.

Designing for Ajax

By studying existing Ajax applications, as well as any relevant precursors, it's been possible to distill a number of important design principles for Ajax, which are shown in this section. The thinking behind the principles was a big influence on the pattern discovery process, and knowing them will help to apply the patterns.

We'll begin by looking at principles of user-centered design, followed by those of software design. Of course, you can never fully separate those concerns, and they're often in conflict with each other. Dealing with those conflicts is really a key concern of the patterns. Incidentally, it's worth checking out a good online resource that takes the opposite perspective: Ajax Mistakes (<http://swik.net/Ajax/Ajax+Mistakes>) is a long list of Ajax mistakes and gotchas, as well as anti-patterns originally authored by Alex Bosworth and now maintained on a wiki.

Usability Principles

Follow web standards

Try hard enough, and you can do some very confusing things with Ajax, even more so as rich graphics become commonplace. Rather than reinventing the Web as we know it, use Ajax to build a "better Web," an enhanced layer over what's already there.

Respect the conventions that users are already familiar with.

The browser is not a desktop

Further to the previous principle, Ajax is a richer brand of the traditional web site rather than a webified brand of the traditional desktop. True, desktop widgets like sliders are migrating towards Ajax, but only when they make sense in a web context and often in a modified form. We're also seeing application categories like word processors head online as well, but again, the best products will be those that fit in with the Web rather than blindly replicating the desktop experience.

If it's different, make it really different

Subtle differences confuse people. If you've decided it makes sense to diverge from a standard or a common idiom, ensure your design is distinct enough to avoid confusion.

Provide affordances

Affordances (<http://en.wikipedia.org/wiki/Affordance>) are as important as ever in Ajax. You may have a fancy new drag-and-drop technique for updating a form, but will the user even know it's possible? Visual design, dynamic icons, and status areas all help here.

Smooth, continuous interaction

Avoid the start-stop rhythm of conventional web apps. Full page refreshes are a distraction and a time-waster. If used at all, they should be reserved for significant, infrequent activities such as navigating to a conceptually new place or submitting a large form.

Customization

Application preferences haven't been very important on the Web. Why bother personalizing background colors for a shopping site you use once a month, especially if it means sitting through a tedious sequence of form submissions? But for some web applications, the user might be spending eight hours a day working with them, and customization suddenly feels a whole lot more useful. With Ajax driving the customization process, it's a lot easier too.

Make it fun

Ajax makes the Web a lot more fun than it used to be. Techniques like visual effects, drag-and-drop, and periodic updates are sometimes labelled "gimmicks" or "eye candy," as if users don't actually enjoy those things. Applied with care and ideally, user testing, they can definitely add real value, even in "serious" applications.

Software Design Principles

Embrace JavaScript

Thanks mostly to the interest in Ajax, JavaScript (<http://www.crockford.com/javascript/javascript.html>) is no longer seen as a basketcase language to be avoided at all costs. It can actually be surprisingly powerful (cut the sniggering already!), provided that developers are willing to study the idioms, patterns, and idiosyncrasies.

Accept workarounds where necessary

Because Ajax is firmly based on standard browser facilities, there's simply no getting around the many constraints imposed by modern browsers. If you seek the benefits of a rich web application that will run immediately on any modern browser, and you consider usability to be critical, then you can only use whatever hacks are necessary. You might well lament that Ajax development is inherently troublesome and pine for a cleaner way to get the job done (I know I do). But the point is that you don't get a lot of say in what the vast majority of the world is already using, and will continue to use in the next few years. Maybe web development *will* be neater one day, but here in the present, the land of reality, it's advisable to deal with the technology at hand and work around—or exploit—whatever cheap tricks are available if they ultimately enable useful functionality that would otherwise be unachievable.

Tame asynchrony

The browser-server communication of an Ajax App is asynchronous by nature. This leads to several risks: users might not be told that a call has failed or timed out; calls might be processed in a nonatomic manner; or the browser might forget why a call was made by the time its response has arrived. As explained in several patterns, there are techniques available to monitor and control calls so that these situations don't arise.

Develop for compatibility

Where JavaScript programming still has issues is in portability. At a syntax level, JavaScript is reasonably consistent across browsers, as the ECMA standardization process (an effort to define JavaScript standards; see <http://en.wikipedia.org/wiki/ECMAScript>) has generally been respected by all the major browser makers. However, it's still a moving target, so older browsers of any variety simply won't support some features. Moreover, the DOM remains a serious portability concern. Despite gradual improvements over the years, there remain many subtle differences, and we're set for more divergence in the future. Developing for compatibility means being explicit about which versions are targeted, using portable libraries where available, and architecting so that portability concerns are isolated from core logic.

Reduce bandwidth

If there's frequent network activity, you'll want to think carefully about the size of the messages being passed back and forth.

Deal with latency

When people talk about "how fast" their connection is, they're usually discussing throughput rate; e.g., "a 4 megabit connection." That's great for downloading large content, but what about interactivity? Latency—basically the time for a bit to travel between browser and server—is usually more important than throughput rate (<http://richui.blogspot.com/2005/09/ajax-reducing-latency-with-cdn.html>). In web apps where the server may be halfway across the world, you can't reasonably respond to each key-stroke or mouse movement due to latency overheads. The challenge is to make the application feel responsive while reducing the frequency of interactions. Techniques like *Submission Throttling* and *Predictive Fetch* make the trade-off by decreasing frequency but increasing the amount of data sent each time.

Partition into multiple tiers

As in any web architecture, Ajax applications should use multiple tiers to help separate concerns. This advice is commonly misinterpreted as implying "keep the presentation simple," a mistake that unfortunately leads to pathetically barren user interfaces. Don't be afraid to create rich, intelligent user interfaces with JavaScript; just be sure to develop for compatibility and ensure your business logic is separate from your presentation logic. In addition, practice "unobtrusive JavaScript" and "unobtrusive CSS." That is, keep your initial HTML page clean by referencing external JavaScript and CSS files; no embedded JavaScript or CSS. JavaScript and CSS should be kept apart too; where possible, the script should change an element's display by switching its CSS class rather than micromanaging style information directly.

Go easy on the browser

Unfortunately, your Ajax App will probably end up being one of many things running in the client machine. This is exacerbated by the fact that JavaScript is pretty slow anyway, meaning that you have to exercise restraint in how much happens in the browser.

Practice graceful degradation

Where a browser doesn't support advanced features, Ajax applications should gracefully fall back to use whatever *is* possible. In an ideal world, the same functionality would still be available, albeit with less bells and whistles. But even if you need to sacrifice functionality—and you often will—it should be done in a graceful manner. (Think “polite error messages” and not “complicated stacktraces.”)

Ajax Patterns Overview

The Ajax Patterns show how people have used the design principles effectively in real-world Ajax applications. It might seem funny that we can have so many patterns about Ajax, a term that was coined only a few months before work on these patterns began. However, the ideas are not new; there were many Ajax features on the Web before the term came about to describe them. The healthy Net economy has helped a lot too, with hundreds of new sites now using Ajax, along with powerful tools (RSS, Technorati, Google, and wikis) to locate them as soon as they're available.

With over 60 patterns, it's useful to classify the patterns hierarchically. At a high level, the book is divided into four parts, each corresponding to a different focus area—Foundational Technology, Programming, Functionality and Usability, and Development. Beyond that, each part is divided into several chapters, where each chapter includes related patterns. For instance, Foundational Technology Patterns (Part II), includes Web Remoting (Chapter 6), which includes several patterns for web remoting. Here's a summary of each part:

Foundational Technology patterns (11 patterns)

The foundational technologies are the building blocks that differentiate Ajax from conventional approaches, and this section explains typical usage.

Programming patterns (23 patterns)

These are the features of architecture and code that serve the software design principles listed previously. These include, among other things, design of web services; managing information flow between browser and server; populating the DOM when a response arrives; and optimizing performance.

Functionality and Usability patterns (28 patterns)

These are the things that matter to users, including widgets and interaction techniques; structuring and maintaining what's on the page; visual effects; and functionality that Ajax makes possible.

Development patterns (8 patterns)

These are process patterns advising on best practices for development, as opposed to all the previous patterns, which are “things” that live inside an Ajax application. The practices are about diagnosing problems and running tests.

Figure 3-1 shows where the four parts sit in the context of an Ajax application. Most patterns—those in the first three parts—are about *the product*, while the remaining part, Development patterns, is about *the process*. Of the product-oriented patterns, the Foundational Technologies explain how to use the crude web technologies such as XMLHttpRequest and the DOM. At a medium level are the Programming patterns, guiding on strategies to use these technologies. At a high level are the Functionality and Usability patterns. Overall, the Foundational Technology patterns are at the core of the Ajax Patterns language; the remaining three parts all build on these, and are fairly independent from one other.

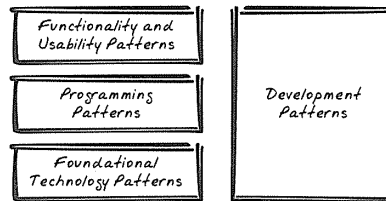
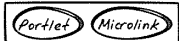


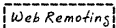
FIGURE 3-1. Ajax Patterns: four parts

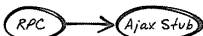
On the inside covers of this book, you'll find an alphabetically ordered reference of all the patterns, providing summaries and page numbers. The introduction to each part of the book and to each chapter also contains some summary information.

In addition, the following pages contain pattern maps for each of the four high-level groups—Foundational Technologies, Programming, Functionality and Usability, and Development. The diagrams in Figures 3-2 through 3-5 follow these conventions.

 Pattern

 Pattern group

 Foundational Technology pattern group

 Pattern A "leads" to Pattern B

 Patterns A and B solve a similar problem, but in different ways

Anatomy of a Pattern

All of the patterns follow the same basic format, though certain fields are left out where appropriate. This section explains the meaning of each section.

Evidence

How much real-world evidence exists for the pattern, on a 3-point scale and presented graphically? It's a rather subjective estimate, but use the following icons as a guide:

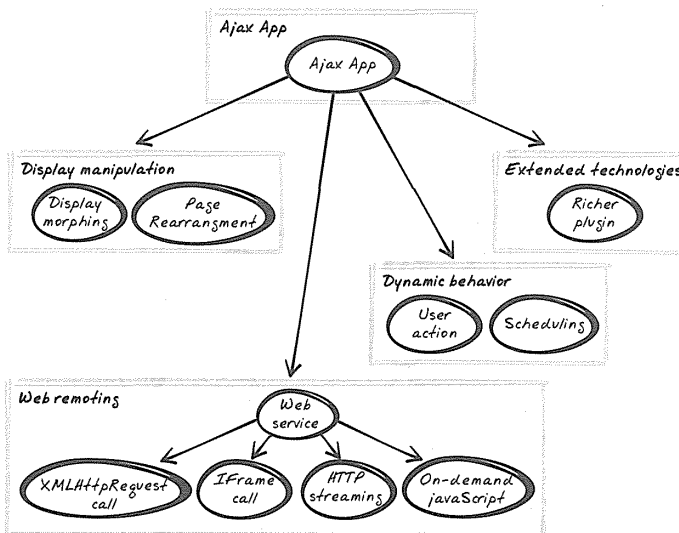


FIGURE 3-2. Foundational Technology patterns



Suggests the idea is purely speculative



Suggests there's at least a proof-of-concept or an early usage



Suggests there's a few established examples



Suggests the pattern is in widespread usage

Tags

Tags—or keywords—help people locate the pattern and get a sense of its focus.

In a Blink

This is a sketch to set the scene for the pattern.

Goal Story, Developer Story

A story is a typical scenario to explain how the pattern is used or what the benefit will be to end users. This pattern has been implemented. Each story is based on a “persona”—a fictitious person (http://www.evolt.org/article/Practical_Persona_Creation/4090/56111/)—to make the story more realistic. (And quite frankly, talking about specific people lets me say “he” and “she” instead of obfuscating the text with gender-neutral language!)

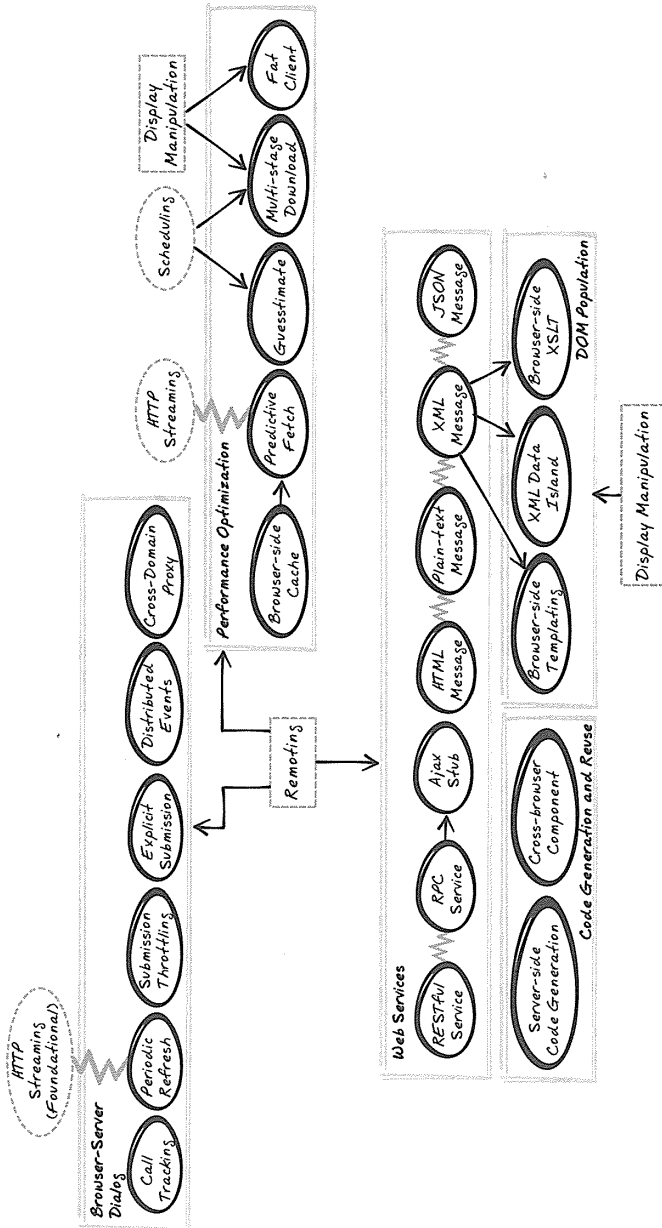


FIGURE 3-3. Programming patterns

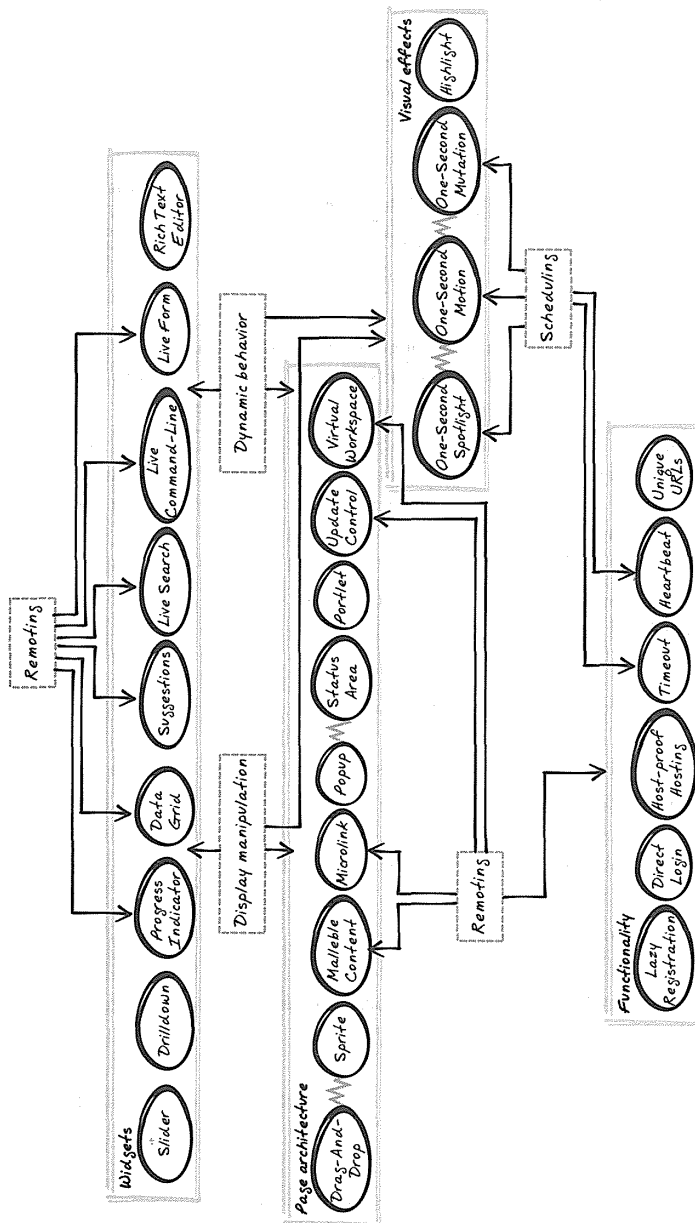


FIGURE 3-4. Functionality and Usability patterns

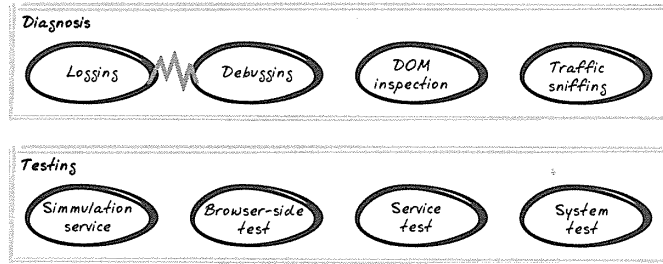


FIGURE 3-5. Development patterns

A small cast of personae is used throughout the patterns. Their names are mnemonic in that they reflect the personae's roles. It's cheesy, but it will hopefully help you remember what these people do without having to refer back here.

The stories in Foundational Technology Patterns (Part II) and Functionality and Usability Patterns (Part IV) are all "Goal Stories"; they illustrate how users interact with a system once the pattern has been implemented. Here are the personae used in all the Goal Stories:

Bill

A Bill-paying citizen with a wife, 2.4 kids, and a dog named after a contemporary sitcom character.

Doc

A Doctor with geeky tendencies.

Frank

A Factory Floor Manager, often stressing about machine safety and worker productivity.

Pam

A Project Manager for a perpetually overdue IT project.

Reta

A Retailer of high-class fashions.

Sasha

A Socialite with plenty of time for social bookmarking, social blogging, and social tagging.

Stuart

A Student with plenty of time for music and other hobbies not related in any way to his studies.

Tracy

A fast-paced financial Trader, dealing in any asset type where there's a buck to be made.

Then there are the stories in Development Patterns (Part V), which show how a Developer uses a particular pattern. These are present in all the “Programming” and “Development” patterns, reflecting the nature of those sections. There are two aptly named developers who appear throughout these stories:

Dave

A senior Developer, skilled in the various Ajax-related technologies.

Devi

A senior Developer, also skilled in the various Ajax-related technologies.

Problem

The problem that is being addressed.

Forces

The forces that arise in addressing the problem.

Solution

A brief solution statement (the first sentence of the section), followed by an elaboration of the technique.

Decisions

Each decision is posed as a question. The decisions are *not* a FAQ section to help clarify the solution—such material would belong in the solution itself. Instead, they are “reusable decisions”—that is, decisions that will often arise once you decide to incorporate this pattern. There is no precise answer given, because each decision must be made pragmatically. The description can only guide you in making the decision, first and foremost by flagging that the decision is there to be made. And beyond that, by alerting you to the variables involved and the consequences of going in one direction or the other.

Real-World Examples

Real-world examples of the pattern at work. Where real-world evidence is lacking, proof-of-concepts and libraries are used instead. Some examples may not be strictly Ajaxian but are included to illustrate a particular point.

Code Example: Refactoring Illustration

A code example and walkthrough.

Where the system is an Ajax Patterns demo rather than an external application, it’s usually a “Refactoring Illustration” because an earlier version is being refactored according to the given pattern. This is explained further at the end of this chapter. Note that Martin Fowler originally defined “refactoring” as a code change without any user-observable behavior change. I am using the broader definition, now in common industry parlance, of a small, incremental improvement, which may well be externally visible.

Alternatives

Other patterns that solve the same problem in a different way. These are often, but not always, Ajax Patterns documented elsewhere in this book.

Related Patterns

Other Ajax Patterns that are somehow related to this pattern, other than the alternative patterns that would have appeared in the previous section. Usually, this means the Related Pattern is a possible follow-up pattern. It may also mean the patterns are somehow similar; e.g., they rely on the same technology.

Metaphor

A metaphor to help remember the pattern. Some people will find this annoying (and should therefore skip over) and others might find it a helpful way to firm up their understanding of the pattern and help remember it later on.

Want to Know More?

Links to any original references and other useful material.

Acknowledgments

Most patterns in the collection were not just speculations, but were discovered from what others were actually doing. As well as the examples above and the initial Acknowledgments section, this section is a place where people can be acknowledged for their contributions.

Ajax Patterns Demos

The Ajax Patterns Demos appear in many patterns as Refactoring Illustrations and also in the Solution section. They're all online at <http://ajaxify.com/run/>, and it would be worth trying them out. In addition, the full code base can be downloaded from that location; its installation is explained in the Appendix B. It includes the demos as well as completed code for Chapter 2. Note the demos have been tested on Firefox 1.5 and IE6, though most of them work on comparable browsers as well.

The server-side code is all PHP, but most of it is fairly trivial, so it should be fairly easy for all programmers to comprehend. PHP was chosen as it's quite easy for anyone to set up, on any platform. Also, the language itself is fairly "generic" in that anyone with web development experience should have no difficulty following the code.

The demos are organized around a refactoring theme. For most of the examples, there is an initial, "embryonic," Ajaxian application. Then, several parallel refactorings are applied to the same demo, each in a separate subdirectory. And each of these refactorings may have a further refactoring applied, contained in a deeper subdirectory. A tree structure emerges on the site, with each application having evolved in different ways.

For example, look at the evolution of Finite Cache Sum Demo (<http://www.ajaxify.com/run/sum/xml/cached/expiry/>), which has the path `/sum/xml/cached/expiry/`.

/sum/

First, there is the basic demo at `/sum/`. Enter some numbers and the server responds with a sum. As a basic Ajaxian application—with no form submission involved—there are some foundational technologies illustrated here, but that's about it.

/sum/xml/

Next, the sum is refactored to receive results in XML form, as a demo is done for the *XML Message* pattern. The “xml” prefix on the URL tells you its lineage.

/sum/xml/cache

One benefit of the new XML form is its convenience for caching, so a further evolution is a basic cache. This is a refactoring illustration for the *Browser-Side Cache* pattern.

/sum/xml/cache/expiry

Finally, the cache undergoes further improvement. This time, a concession to the laws of physics is made, and the cache is now of finite size. Unused elements expire. So here’s a further refactoring illustration for *Browser-Side Cache*.

Conclusions

This chapter has covered how people are designing for Ajax and introduced the Ajax Patterns. The pattern language (just like Ajax itself) isn’t a magic bullet, but a tool intended to improve your mastery of Ajax web development. You can use it as a reference for “quick-fix” problem solving, but you probably gain more if you also treat it as an educational resource to learn more about the recurring problems and solutions in Ajax. The remainder of this book constitutes the patterns themselves, divided into four parts according to their area of concern—Foundational Technologies, Programming, Functionality and Usability, and Development.

Foundational Technology Patterns

THE FOUNDATIONAL TECHNOLOGY PATTERNS OUTLINE THE “BUILDING BLOCKS” AT THE HEART OF ANY Ajax application. The patterns are considered atomic within the pattern language, in the sense that all later patterns build on these basic techniques. To keep the Ajax Patterns concise, you will only encounter technologies that add something to conventional web development; there’s no “Form Submission” pattern, for instance.

Ajax itself is a pattern, hence the first pattern Ajax App (Chapter 4), which acts as the root for all the Ajax patterns. The next few chapters look at the three core technologies involved in creating an Ajax App. Display Manipulation (Chapter 5) patterns are about repainting the user interface. Web Remoting (Chapter 6) patterns document several alternatives for communicating with the server. The Dynamic Behavior (Chapter 7) patterns are about events—those derived by users and those based on timing. Finally, Extended Technologies (Chapter 8) explains how you can extend an Ajax App with nonstandard technologies.

Not all of these are patterns in a conventional sense—some might argue that a pattern like *XMLHttpRequest Call* is simply an overview of a technology. Personally, I do see them as patterns insofar as they are good, proven ways to solve recurring problems—it’s just that the problems arise in different contexts and might be applied by different people (such as someone sketching out a potential project or setting up its overall architecture). So while the patterns here do have a different quality, I find the pattern form is a suitable way to document these things, and I find that these foundational technologies—whether you prefer to call them patterns or not—fit nicely into the overall pattern language.

CHAPTER FOUR

Ajax App

T

HIS CHAPTER CONTAINS JUST A SINGLE PATTERN, THE ROOT FOR THE ENTIRE PATTERN LANGUAGE:

Ajax App.

Ajax App



Ajax, Balanced, Client-SOA, DHTML, Fast, Fat, Interactive, Platform, RichInternetApplication, Rich, Thick, Web2.0, and WebApp

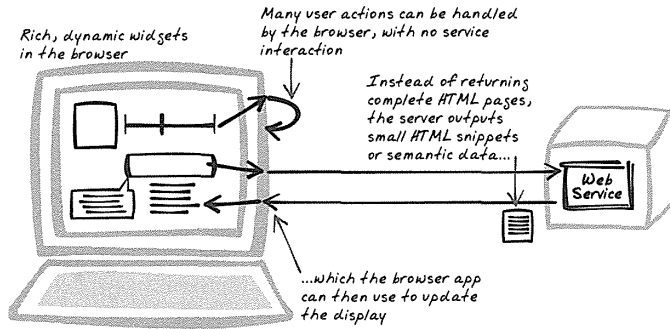


FIGURE 4-1. Ajax App

Goal Story

Pam's begun entering staff appraisals into a new Ajax App. She's pleased to see the data entry is much faster: fields are validated as soon as they're filled out, searches are integrated into the form rather than in annoying popups, and the remaining form fields keep mutating to reflect what she's entered so far.

Problem

How can you create a rich application?

Forces

See Chapter 1 for more details on the forces driving Ajax, which are summarized here.

- Users like working and playing in the browser and are now more willing to keep their data online, but are nonetheless frustrated with conventional "click 'n' wait" interfaces.
- Companies like web apps running on their Intranets because it makes deployment much easier, but they continue to be burned by unusable web apps that don't deliver the same value as a comparable desktop app.
- Developers are now well-versed in the basic patterns of web architecture and ready to take on new challenges.
- Technology has opened up new opportunities for the Web: broadband and beyond is becoming ubiquitous in many countries; servers can process huge quantities of requests per second; and storage is growing to the point where it's feasible for individuals to host most of their personal data online.

Solution

Produce an Ajax App—a rich application capable of running inside any modern web browser. The user fires up an Ajax App by typing in a URL, clicking a link, or

activating a bookmark. The Ajax App is then downloaded into the browser and the user can begin working with it immediately.

Ajax is fundamentally a user-centered pattern. As mentioned in the first chapter, the major force driving Ajax is the needs of users and their organizations to get more out of the Web. For too long, people have endured pages that take forever to load, widgets that make no sense, and forms that conveniently “forget” their data when they take too long to fill out. There are good reasons for people to continue using a browser, as noted in Chapter 1, but it’s in the interests of web site owners and employers to do a better job at making that interaction productive and enjoyable.

Ajax aims to improve user experience within the constraint of respecting browser standards. Standard-based design obviously has the benefit of portability, but also has a usability component in itself; users interact with hundreds of web sites and appreciate consistency, a quality that quickly erodes when you rely on plugins, browser-specific features, and “creative” design ideas. Sure, incorporate a novel widget or a funky layout when it works, but for most things, the standard way works best.

Non-standard technologies also break consistency; one reason Flickr (<http://flickr.com>) began migrating from Flash to Ajax was because users want to right-click to save images, a standard broken by Flash (http://www.ajaxian.com/archives/2005/05/ajax_summit_eri.htm). Again, don’t be afraid to use a technology like Flash to achieve something standard browsers won’t do—that’s the whole idea behind the *Richer Plugin* (Chapter 8) pattern. Some people will say it’s not an Ajax App if it uses Flash, but the definition here would include a mostly Ajax App that delegates to Flash (or other plugins) to fulfill some specialized task. In any event, the definition really isn’t worth arguing about (after all, you can define “Ajax” however you like); the important thing is to ask what helps users—sometimes, it’s better to incorporate some Flash or Java; other times, a desktop solution really is the best approach.

The standard technologies are discussed in Chapter 1. They include several well-established technologies: HTML, CSS, HTTP, CGI, and form submission, and server-side scripting. In addition, there are several more technologies that are only now becoming seriously popular and standardized in the browser: JavaScript, the Document Object Model (DOM), the Event Model (DHTML), and Web Remoting.

Decisions

Will your application be “Ajax Deluxe” or “Ajax Lite”?

There are two archetypal architectures for Ajax, and all applications lie somewhere along the spectrum between these extremities.*

* Harry Fuecks explains essentially the same distinction as “Client/SOA” versus “HTTP++” (<http://www.sitepoint.com/blogs/2006/02/15/two-kinds-of-ajax-html-vs-client-soa/>) Likewise, Craig Shoemaker contrasts “Ajax Applications” with “Ajax Websites.”

Ajax Deluxe

Ajax Deluxe uses Ajax to the max: applications feel similar to a desktop in that the browser is driving the interaction—it mediates all interaction between the user and server, so there are no—or few—direct page refreshes. Similarly, there’s no need for session tracking in the server, because in the absence of page refreshes, all relevant state can be retained inside the browser script. The server need not know about HTML at all, and might just offer generic *Web Services*. An example is the Ajax calendar, Kiko (<http://kiko.com>).

Ajax Lite

An Ajax Lite App feels more like a conventional web app overall, but one that’s been sprinkled with Ajax here and there. For instance, Ajax might be used to validate a form before it’s submitted using standard form submission, or it might be used to reveal some instructions on the page when a user requests help. An example is Delicious (<http://del.icio.us>), which works fine on legacy browsers, but offers the Ajax *Suggestions* (Chapter 14) pattern for the many browsers that support it.

Which will you use? The Deluxe approach suits a development team with more advanced web programming knowledge and access to relevant tools and cross-browser libraries, and generally leads to a nicer, more effective user interface. It also facilitates a well-partitioned architecture, since the presentation logic can be completely isolated inside the browser, and the business logic completely isolated in the server. However, Deluxe applications may place a strain on the browser and network capabilities, and might not even be possible if the browser is outdated. Ajax Lite is a better answer for older browsers, since the Ajax features can usually be “turned off” to support graceful degradation.

What browser requirements will there be to run your Ajax App?

One of the reasons why Ajax has taken off now, in spite of people having tried similar things earlier on, is that browsers are now more consistent. For an Ajax App, you’ll need to decide which browsers you’re targeting and which browsers just aren’t worth the effort. XMLHttpRequest is the main constraint, because it usually appears later in a browser’s feature set than other foundational technologies. Roughly, we’re looking at browsers from around 2001 onwards. A typical guideline is: Internet Explorer 5+ for Windows, Firefox 1+, Mozilla 1+, Safari 1.2+, or Opera 7.6+ (http://en.wikipedia.org/wiki/Ajax_%28programming%29). Other browsers such as Netscape and Konqueror might be considered too. Whether you’ll support all these browsers, or just a subset, depends on how important the diversity is, how much expertise or library support you have, and how much time you can devote to testing.

How will you treat browsers with insufficient technologies for Ajax?

Building on the previous point, what happens when your application requires features not provided by the browser being used. As explained in *Cross-Browser Component* (Chapter 12), you can either check for specific features or you can check browser versions. If you do the former, you have the opportunity for graceful degradation—to cut out one specific feature

or provide a simpler alternative. It's more work, but lets you support more users. The alternative is to give up and suggest the user upgrade browser, sometimes the only thing you can reasonably do.

How much processing power will be required of the browser?

A few early Ajax Apps have already been bitten by assuming too much of the client-side environment. Unfortunately, JavaScript doesn't execute too quickly, especially when the user has lots of browser windows open, as well as a music player, mail client, and the usual array of resident background apps. There are no well-established guidelines here, so the main thing to do is simulate the kind of environments your app will run in, and test, test, test.

How much networking capacity will be required?

With Ajax Apps, there are two networking factors to consider:

Throughput (bandwidth)

How much data can be transferred per second?

Latency

How much time between a packet leaving the browser and arriving at the server (or vice versa)?

In some cases, Ajax actually reduces throughput requirements by avoiding full page refresh. Yet, it's important to consider, because there can still be a strain caused by patterns like *Periodic Refresh* (Chapter 10), which continuously pulls down new data. When you consider that some Ajax features (e.g., *Live Search* [Chapter 14]) require round trips to the server on just about every keystroke, you can see why latency is becoming a growing theme in web design. For smaller packets, which are common in Ajax Apps, latency is actually a much greater source of delay than throughput, especially across long distances (<http://richui.blogspot.com/2005/09/ajax-reducing-latency-with-cdn.html>). Again, it comes down to benchmarking and deciding what's feasible. If a round trip takes several hundred milliseconds, responding to each keystroke will probably cause more frustration than joy.

Real-World Examples

Please refer to the examples in Chapter 1.

Code Example

Please refer to the code examples in Chapter 2.

Alternatives

Conventional web app

Ajax represents an evolution from the conventional web app, where interactivity was limited to form-based interaction with a little JavaScript enhancement, such as client-side validation. Over conventional web apps, Ajax has the following advantages:

- Less waiting time, which makes users more productive and less frustrated.
- The browser can be kept up-to-date without the user having to keep hitting refresh.
- Superior widgets make input more expressive.
- Visual effects and dynamic browser activity make the application feel more alive and give users a better feeling of control.

Conventional web apps do have some advantages:

- Users are more familiar with conventional apps.
- Developers don't need to learn new techniques.
- They are compatible with older and non-standard browsers.

Flash app

As pointed out earlier in the "Solution," a standard web app that uses some Flash here and there can still be considered an Ajax App. Here, though, we're comparing Ajax to full-blown Flash, which creates the kind of applications that won't do anything unless Flash is enabled.

Flash and Ajax are actually a close match. Google Maps (<http://maps.google.com>) caused surprise when it came out, as many assumed that only Flash could produce something as rich inside the browser. Yahoo! then turned it full circle by producing a similar product, Yahoo! Maps (<http://maps.yahoo.com/>), in Flash. Despite the similarities, significant differences do remain. Compared to Flash, Ajax has the following benefits:

- Ajax is more portable as it runs on any standard browser, although Flash support is still quite widespread.
- Ajax Apps will often be more consistent, using standard web conventions familiar to most users. Flash has less established conventions, so using an App is somewhat less predictable.
- Ajax works better with nonbrowser clients such as search engine robots.
- Ajax plays nicer with browser settings such as color preferences, and also with browser-specific features; e.g., skins, autocompletion, and Greasemonkey-style enhancements.
- Ajax is safer long-term as there's no lock-in to a proprietary plugin.
- Due to its open nature, the community of Ajax developers is growing rapidly, so that libraries, techniques, and (cough) patterns are evolving rapidly.

Following are the benefits of Flash over Ajax. Whether these are compelling enough to write a pure Flash app depends on the circumstances; keep in mind that the most appropriate solution is often a combination of Flash and Ajax when it's not possible to stick with pure Ajax.

- Flash offers richer graphics and video.
- Flash allows for sound effects and playback of sound files, as well as audio input.
- Rich graphical programming may be easier in Flash because, unlike Ajax, it's designed with those kind of applications in mind.
- Flash allows for local storage.

Java applets

For some Java developers, Ajax is déjà vu. In the mid-to-late '90s, Java applets were destined to take over the browser and turn the desktop into a trivial sideshow. It didn't happen for various reasons, and Java applets are now rarely seen on the Web. Still, they persist in the enterprise, and as with Flash, it's possible to combine the technologies. Some have speculated that all the interest in Ajax might breathe some life into the world of applets; it doesn't seem to be the case so far, but it's too early to be sure.

Ajax has several advantages over Java applets:

- Ajax Apps load in a matter of seconds, where Java applets take longer because they tend to use larger libraries and have to load them all to get started.
- Ajax Apps look and feel like standard web apps, while Java applets feel more like desktop apps and somewhat out of place in the browser.
- Ajax features can grow organically from a conventional web app, whereas Java uses a completely different programming style that has to be written from scratch.
- Java versions have changed over the years, and there are often incompatibilities between an applet and the Java environment used by the browser. Furthermore, you usually have to lag a few years behind the most recent version of Java.

Java applets have several advantages over Ajax:

- Java is often used on the server side, so there are synergies in a 100 percent Java solution, such as shared code and the ease with which developers can transcend tiers.
- Many people know Java already, and an applet architecture means they virtually don't have to know anything at all about web technologies.
- There is good support for desktop-like user interfaces, if that's what is desired.

Sun has more recently introduced Java Web Start as a cross between the desktop and the browser. Web Start apps run off the desktop, but can be launched by activating a URL in the browser. The huge waiting times of applets are avoided because libraries can be held locally and shared across applications. It's a useful alternative to applets and more geared toward the enterprise, though it also hasn't taken off in a big way.

Desktop app

There has always been a decision to make between the desktop and the Web. Ajax ramps up the web argument, with even mainstays of the desktop, like word processors and spreadsheets, becoming commonplace in the browser. Microsoft's "Live" strategy, which involves Ajaxified versions of its Office apps, is a major case in point. Still, other applications like games and multimedia tools won't be budging from the desktop for a long time to come.

Ajax offers the following advantages over the desktop:

- An Ajax App doesn't need to be installed; it's available on any computer with a browser. Thus, it suits people who use multiple computers and people who use computers that they can't control.
- Ajax data is hosted on the network, so users can access it from anywhere, and it can be backed up automatically. While this is feasible with desktop apps too, it doesn't tend to happen that way for several reasons; e.g., data migration would be difficult because there are so many different desktop versions running at the same time.
- Most users are comfortable with web interaction mechanisms, and well-designed Ajax Apps exploit that familiarity.
- Ajax works well with the Web. Users can bookmark a "position" within an Ajax App, click on a hyperlink within the Ajax App, and so on.
- Ajax Apps are upgraded with no user intervention.
- Developers can choose the server-side programming and runtime environment according to their own needs, whereas desktop apps impose many constraints.

Ajax has bridged the gap, but desktop apps can still do many things that are beyond the realm of standard web browsers (see also *Richer Plugin* [Chapter 8]):

- Provide local file access.
- Use sound, taking full advantage of modern sound cards.
- Use rich graphics, taking full advantage of modern video cards.
- Use keyboard shortcuts, where support is varied in the browser.
- Provide hardware access.
- Provide communication to machines other than the base server, and in protocols other than HTTP.
- Provide OS-specific interaction, such as popping up alerts.
- Offer faster processing, since JavaScript running in a browser is much, much slower than executable machine code running on standalone.

Mobile app

With mobile apps growing in parallel with Ajax, it's worth considering them as an alternative, albeit a less direct one. As it happens, Ajax itself may actually become a good platform for mobile apps, with Opera now providing Ajax support on its mobile platform (<http://www.opera.com/products/mobile/platform/>).

Related Patterns

All of the patterns in this language relate in some way to Ajax App. In particular, the earlier "Solution" points to the Foundational Technology patterns that really define what Ajax is all about.

Display Manipulation

FOR AN END USER, THE MOST OBVIOUS THING ABOUT AJAX IS ITS VISUAL APPEARANCE. AJAX APPS TEND to look richer, and the interface tends to update smoothly, more like a desktop app than a conventional web app. This chapter describes the two main technologies for updating the user interface. *Display Morphing* focuses on changes to a single element, while *Page Rearrangement* involves changes to the page structure. Along the way, we'll look at the Document Object Model (DOM). DOM manipulation is key to many of the Ajax Patterns.

Display Morphing



Display, DOM, Graphics, GUI, Morph, Page, Paint, Reference, Rich

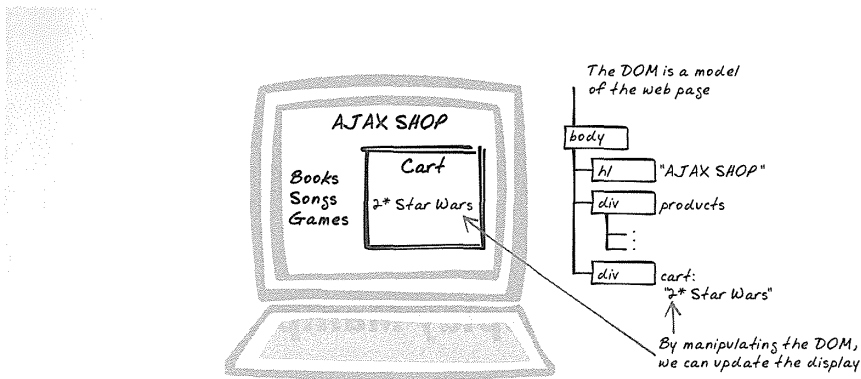


FIGURE 5-1. Display Morphing

Goal Story

Stuart is answering a question in an online quiz, with a countdown box showing how much time remains. The countdown label changes each second, and the color gradually shifts from green to red as the countdown proceeds.

Problem

How can you dynamically update display elements?

Forces

- As the user's task and context changes, applications need to present different types of information.
- As information the user is working on changes, information being displayed becomes stale.
- Stale information is a big problem in conventional web apps. Because the information cannot change until it's out of date, you sometimes see disclaimers such as "May Be Out of Date" or instructions like "Click to Refresh."
- Submitting and redrawing the entire page interrupts the user's flow.

Solution

Morph page elements by altering styles and values in the Document Object Model (DOM), such as text and color properties. The DOM represents the state of the web page at any time. When JavaScript manipulates the DOM, the browser notices the changes and immediately reflects them in the user interface. To morph a page element's display, you get a reference to it and change its properties. This allows you to change the display of any element on the page—headings, div elements, images, and even the document object itself. Note: an online demo (<http://ajaxify.com/run/display>) illustrates the code concepts throughout this Solution and the code snippets loosely follow from the demo.

To further illustrate the DOM, consider the following piece of HTML:

```
<div id="display">
  
  <div id="dimensions">
    <input id="width" name="width" type="text" size="20"/>
    <input id="height" name="height" type="text" size="20"/>
  </div>
</div>
```

Figure 5-2 shows the structure that arises when the browser transforms the HTML string into a DOM model. Since each node has attributes, we can represent the model in more detail, as shown in Figure 5-3. To learn more about the DOM, experiment with the tools mentioned in *DOM Inspection* (Chapter 18). You can point them at a web page to view the corresponding DOM model.

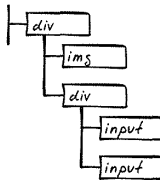


FIGURE 5-2. DOM structure

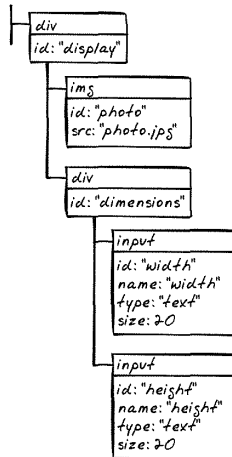


FIGURE 5-3. DOM structure with attributes

Each node has a number of standard attributes and operations. Those below let you inspect surrounding DOM context:

`parentNode`

The element's parent node.

`childNodes[]`

An array of all immediate child nodes.

`getElementsByTagName("tagName")`

An array of all child nodes of type `tagName` (e.g., `h1` for all level-1 headings).

The structure can be manipulated as well as inspected. Following are the key manipulation methods available on all nodes (discussed further in *Page Rearrangement*):

`appendChild(newChild)`

Appends `newChild`.

`insertBefore(newChild, oldChild)`

Inserts `newChild` just before `oldChild`.

`removeChild(oldChild)`

Removes `oldChild`.

You can also inspect the node itself; useful properties include:

`id`

The node's ID.

`nodeType`

Usually 1 to denote a standard element. 2 represents a tag attribute and 3 represents a text node (explained later in this section).

`tagName`

The name of the tag (e.g., `"h1"`).

For a more complete set of properties and operations, a good source is available at HowTo-Create.co.uk (<http://www.howtcreate.co.uk/tutorials/index.php?tut=0&part=28>).

Before using any of these properties, the first thing you need to do is get hold of a node somewhere in the DOM. Here's how to grab a reference to an image node with an ID of "logo" buried anywhere inside the DOM:

```
var logo = document.getElementById("logo");
```

`getElementById()` will recurse through the whole document to find the element with a given ID property. Since calls like this are extremely commonplace, a convenience function is used throughout the demos. The function name is just one character, `$`, which leads to a neater syntax like this (as explained in Chapter 2):

```
var logo = $("logo");
```

Now that we have a reference to the image node, we can play around with its properties. To start with, we can change its `src` property, which will cause it to load a new image:

```
logo.src = "Images/web.png";
```

We can also set the image's alt text, which will show up on text browsers, and as a tooltip on some browsers):

```
logo.alt = "The World Wide Web"
```

So far, we've altered a couple of image-specific properties. Each HTML element has its own set of properties that can be manipulated via DOM manipulation. There are references such as the w3schools web site (http://www.w3schools.com/html/dom/dom_reference.asp) that list the properties for each element.

Another common task is manipulating text, which in the DOM is actually represented by special text nodes. This is a little confusing as there's no corresponding HTML tag. Look at the following text, which seems to have some text freely roaming inside a <p> tag:

```
<p>Oh no! It's <strong>Mr. T</strong>!</p>
```

Text always resides inside. This is also a little confusing as there's no corresponding HTML tag; the text nodes are implicit. In the example above, what the DOM holds is actually a paragraph node <p> with three children: a text node, a strong node with a text node inside it, and another text node. It's represented in the DOM as follows:

```
<p>
  <textnode>Oh no! It's </textnode>
  <strong><textnode>Mr. T</textnode></strong>
  <textnode>!</textnode>
</p>
```

Although the initial HTML is trivial, building up a message like this with DOM manipulation is somewhat complex:

```
message.appendChild(document.createTextNode("Oh no! It's "));

var strong = document.createElement("strong");
strong.appendChild(document.createTextNode("Mr. T"));
message.appendChild(strong);

message.appendChild(document.createTextNode("!"));
```

Even when text nodes aren't involved, directly manipulating the DOM can get quite ugly. A popular alternative is to use `innerHTML`, which lets you specify an HTML snippet. Thus, we can rewrite the code above by simply setting the `innerHTML` property:

```
message.innerHTML = "Oh no! It's <strong>Mr. T</strong>!";
```

By setting the element's `innerHTML` property, we've effectively delegated the DOM manipulation to the web browser. It's much simpler and easier to understand than direct DOM manipulation and is supported by all modern browsers. However, use it with caution, because if you give it an invalid HTML string, you might end up with a subtle bug related to an unexpected DOM state. Also, there are some subtle portability issues, which mean that a particular HTML segment won't always produce quite the same DOM model. For instance, certain kinds of whitespace will be captured as a text node by Firefox, but are

then ignored by IE (<http://www.agavegroup.com/?p=32>). So if you use `innerHTML`, be careful about subsequent DOM manipulations on and around that content.

On the whole, direct DOM manipulation is often more appropriate for complex operations. IE also offers an `outerHTML`, less commonly used, which will not only set the element's contents, but also overwrite the element itself. With `outerHTML`, it's as if the element is replaced by the contents of its `outerHTML` property.

If we're looking at changing the display, a particularly important aspect to consider is CSS style. CSS styles and classes are just regular properties and can be manipulated just like the others. Assume the stylesheet defines a CSS class called `inYourFace`. When we change the message's `className` property to `inYourFace`, its display will automatically update to reflect the `inYourFace` definition.

```
[CSS]
.inYourFace {
  padding: 10px;
  background-color: #ff4444;
  font-size: 250%;
}

[Javascript]
message.className = "inYourFace";
```

As explained in the next section, changing appearance by switching classes is a good practice, but there are also times when the JavaScript needs to manipulate style directly via the `style` property present on all elements. `style` itself has a number of properties, as you would see in a stylesheet, but with JavaScript-ish camelCase (`backgroundColor`) like that shown below, as opposed to the CSS hyphenated style (`background-color`):

```
message.style.padding= "10px";
message.style.backgroundColor = "#ff4444";
message.style.fontSize = "250%";
```

Decisions

Will you alter the display via `classname` or `style`?

For those morphings related to CSS properties, you have a choice between manipulation via `style` and `className`. As a general rule of thumb, use `className` when you can and use `style` for special situations. Using `className` follows the principle of unobtrusive JavaScript, because it clears away any mention of fonts, colors, and layout from the code. Thus, the code logic is clearer and the presentation details are encapsulated better inside the stylesheet.

So what are those special cases where *style* should be altered? Here are a few situations:

Style depends on some variable

For example, an HTML histogram, where the height of each element is calculated according to the value it represents.

Animation

Animations are a special case of the previous point, where styles usually vary according to elapsed time.

Multiple variables

Sometimes each style property is to be tied to a particular variable; e.g., font reflects age, color reflects keyboard focus, and so on. There would be too many combinations to hold each as a separate style. You could, however, hold each as a separate class, since an element can have more than one class (its class property can be a whitespace-separated list of class names).

Prototyping

When in creative-coding mode, setting the style directly is sometimes easier than messing around with a separate stylesheet. But don't forget to refactor later on!

What sort of properties will be used?

In modern browsers, the DOM is very powerful, so you can change just about anything you can see. The following are some of the properties that can be typically altered. Note that, as just mentioned, many of the CSS styles will usually be altered indirectly, via `className`.

Color—`style.backgroundColor`, `style.fontColor`

- Change to a distinguishing color to highlight an entire image or control—for example, to draw the user's attention or indicate that an item has been selected.
- Change to a distinguishing color to highlight a range of text. This could be combined with a font color change to indicate that text has been selected.
- Change to a symbolic color to denote some status change; e.g., "red" for stopped, and "green" for running.
- Provide an animation effect by fading or brightening a control. This can draw attention more effectively than can a sudden change in the control itself.
- Change color according to a variable; e.g., the brightness of a blog posting's header varies according to the number of times it's been viewed.

Background Image—`style.backgroundImage`

- Change the image to indicate the status of an element; e.g., a control might be stopped or started, or source code might be OK, have warnings, or have errors.

Border Style—`style.borderWidth`, `style.borderColor`, `style.borderColor`

- Highlight/Select an element to draw attention to it.
- Indicate whether some text is editable or read-only.

Font Style—`style.fontSize`, `style.fontWeight`, `style.fontStyle`

- Change the size, weight, and/or slant of a font to highlight some text.

»

Inner/Outer HTML—style.innerHTML, style.outerHTML

- Change some text content.
- Change arbitrary HTML, possibly changing the nested HTML of an element as well.

Image Source—src

- Change an image object by modifying src to point to another image URL.

Real-World Examples

Ajax-S

Robert Nyman's Ajax-S (<http://www.robertnyman.com/ajax-s/>) is a slideshow manager—think “Powerpoint Lite”—that morphs the slide content as you flick through the presentation (Figure 5-4).

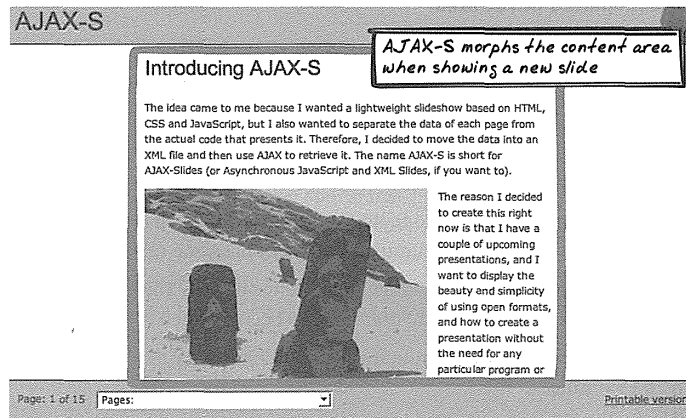


FIGURE 5-4. Ajax-S

Digg Spy

Digg Spy (<http://digg.com/spy>) shows new stories and events such as user moderation as they happen (Figure 5-5). What's interesting from a Display Morphing perspective is the fade effect used to highlight each item as it appears.

Ajax Spell Checker

The Ajax Spell Checker (http://www.broken-notebook.com/spell_checker/index.php) highlights any misspellings. It morphs the words into a red, underlined form that is familiar to any MS-Word user. A similar service is also offered by another Ajax App, Gmail (<http://gmail.com>).

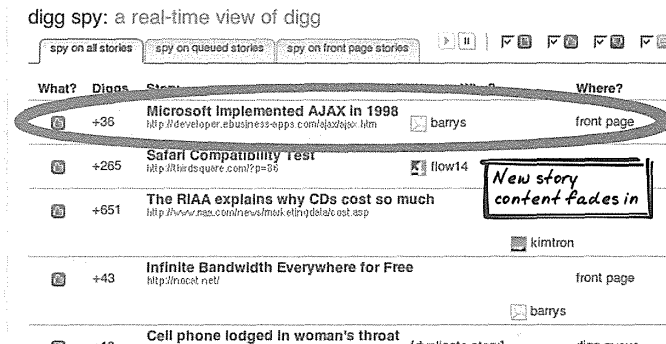


FIGURE 5-5. Digg Spy

Code Example: AjaxPatterns Countdown Demo

The example is a simple countdown demo (<http://www.ajaxify.com/run/countdown/>). A number counts down with the background color indicating “heat level”—green at the start, and transitioning to red at the end.

The div that changes color is called `timeout`. Its initial declaration simply specifies its ID and class:

```
<div id="timeout" class="status"></div>
```

Using techniques covered in the *Scheduling* (Chapter 7) pattern, the startup sequence arranges for `onTick()` to be called every 100 milliseconds. At each interval, `timeout` is modified. `innerHTML` changes to alter the text and `style.backgroundColor` changes to alter the color:

```
$("#timeout").innerHTML = Math.round(secsRemaining);
$("#timeout").style.backgroundColor = calculatePresentColor();
```

The background color is determined by a function that manages the transition from green to red according to `secsRemaining`. Colors are usually represented with hex RGB values—e.g., you can set `style.backgroundColor` to the string, `#00ff00`, to make it green. However, there’s a more convenient notation for this, based on percentages, so an equivalent representation is `rgb(0%,100%,0%)`. The algorithm here gradually increases the red component from 0 percent to 100 percent and decreases the green component from 100 percent to 0 percent. Blue is fixed at 0 percent. At each moment, the red component is assigned to the percentage of time remaining and the green component to the percentage covered so far.

```
function calculatePresentColor() {
    var secsSoFar = TOTAL_SECS - secsRemaining;
    var percentSoFar = Math.round((secsSoFar / TOTAL_SECS) * 100);
    var percentRemaining = Math.round((secsRemaining / TOTAL_SECS) * 100);
    return "rgb(" + percentSoFar + "%, " + percentRemaining + "%, 0%)";
}
```

Related Patterns

Page Rearrangement

Page Rearrangement looks at manipulating the overall structure of the page, whereas this pattern looks at the appearance of individual elements. Together, the two patterns cover all aspects of display manipulation.

Metaphor

Display Morphing is like using a magic paintbrush to change the appearance of anything on the page.

Want To Know More?

- InnerHTML versus DOM Discussion between Alex Russell and Tim Scarfe (<http://www.developer-x.com/content/innerHTML/default.html>)

Page Rearrangement



Add, Adjust, Change, Delete, DOM, Move, Overlay, Rearrange, Restructure, Remove

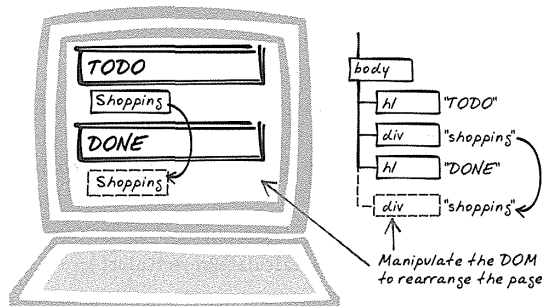


FIGURE 5-6. Page Rearrangement

Goal Story

Doc is looking at a X-ray image in an online diagnostics system. When the mouse rolls over the body, an “advice box” appears at the bottom of the page with stats to help interpret the image. In case the box gets in the way, he’s able to drag it around the page.

Problem

How can you dynamically restructure the page?

Forces

- As the user’s task and context changes, Ajax Apps need to present different information.

- Changes to page structure should be as smooth as possible.
- Refreshing the page breaks continuity and also clears JavaScript state.
- Refreshing the page only allows for a discrete transition from one appearance to another—it's not possible to gradually fade or move an item.

Solution

Add, remove, move, and overlay elements by manipulating the DOM. The DOM is a hierarchical structure that defines how elements relate to one another. By adjusting the DOM structure, you can adjust where elements appear on the page. There are also critical CSS styles that affect page structure—you can change these by manipulating DOM elements' style attributes. Note: an online demo (<http://ajaxify.com/run/page>) illustrates the code concepts throughout this Solution and the code snippets loosely follow from the demo.

Adding is normally achieved by introducing a new element as a child of an existing *container* element, such as `body` or `div`. For example, you can create a new `div` element and add it to a parent `div`:

```
var message = document.createElement("span");
$("container").appendChild(message);
```

Another way to insert an element is to append to the `innerHTML` property of the container element:

```
$("container").innerHTML += "<span></span>";
```

The opposite action is *removing*. By removing an element from the DOM, you take it off the display:

```
$("container").removeChild($("#message"));
```

As a variant of adding and removing, you can keep an element on the page but toggle its visibility using CSS. There are two alternative style properties you can use: the `visibility` style and the `display` style. The former will always preserve layout while the latter will cause the element to squeeze in and out when it's shown or hidden. So with no `visibility`, it's like the element's still there but wearing a coat of invisible paint. This makes layout easy, because everything stays where it is, but it can be ugly to have a big patch of whitespace on the page. With the `display` style, it's more like the element's really disappeared, so the other elements squeeze in to take the place it had occupied. This makes layout a bit more complex, but is usually better visually. The `visibility` property alternates between `visible` and `hidden`:

```
$("#message").style.visibility = "visible"; // Now you see me
$("#message").style.visibility = "hidden"; // Now you don't ("Invisible paint")
```

`display` is actually used for more than just showing and hiding an element—it also defines how an element appears when it *is* visible. Briefly, the main options are `block` (the default

for div elements), inline (the default for span elements), and none (when it's not to be shown).

```
$("#message").style.display = "block"; // Now you see me, with block layout
$("#message").style.display = "inline"; // Now you see me, with inline layout
$("#message").style.display = "none"; // Now you don't ("Gone away")
```

You can *move* an element around in a couple of ways. First, you can remove it from one place in the DOM and add it to another:

```
container.removeChild(message);
extraContainer.appendChild(message);
```

Second, you can adjust CSS properties. The most direct styles are left, right, top, and bottom, which define the coordinates of the element in question:

```
message.style.top = "150px";
message.style.left = "50px";
```

But what is the (0,0) point these coordinates are relative to? The precise meaning of these properties is modulated by the positioning element.

- If static, the coordinates have no effect—the element is positioned according to standard CSS layout rules.
- If relative, the coordinates are relative to the position it would normally be positioned under standard CSS layout rules—they suggest how far its displaced.
- If absolute, the coordinates are relative to the top-left of the entire document.
- If fixed, the coordinates are relative to the top-left of the browser viewport (the portion of the browser window that shows the page). Even if you scroll the document, the element will stick right on the same point on the screen as before.

Positioning is set with standard CSS styles:

```
message.style.left = "150px";
```

Finally, you can also *overlay* elements. An HTML document has “2.5” dimensions, which is to say that it has a limited notion of depth in which elements are able to overlap each other. The critical CSS style here is *zIndex*, which signifies an element's depth. When two elements occupy the same portion of the page, the element that will appear in front is the one with the higher *zIndex*. The *zIndex* is not a real depth, because all that matters is the *relative* ordering of *zIndex* values. Against a *zIndex* of zero, it makes no difference whether an element's *zIndex* is 1, 10, or 100. The default value is 0, and a *zIndex* can take on any positive or negative value.

```
message.style.zIndex = -100; // Behind of all elements with default zIndex
message.style.zIndex = 100; // In front of all elements with default zIndex
```

Decisions

Which positioning style to use?

A single application can combine different types of positioning. In most cases, static—the default positioning—suffices. Nonstatic positioning is most commonly used with more free-floating elements, such as *Sprites* (Chapter 15) or elements suitable for *Drag-And-Drop* (Chapter 15). For nonstatic positioning, relative positioning tends to be the most useful, because it allows you to move the element around within a defined container.

How will you protect against memory leaks?

Continuously adding and removing elements leads to the risk of memory leaks. JavaScript is supposed to perform garbage collection, automatically removing variables that are no longer referenced. However, it's sometimes buggy (notoriously in IE) and, in any event, you have to be sure that elements are really dereferenced when they're no longer used. Some general guidelines are as follows:

Avoid global variables where possible

Local variables go out of scope, so if a local variable points to a deleted element, the element will disappear. But if a global variable points to such an element, it will stick around.

Explicitly nullify references

If you're sure a variable will no longer need to use the value it references, set the variable to null.

Avoid or destroy circular references

You can sometimes end up with a cyclic structure that no one's using anymore, but that sticks in memory because garbage collection isn't smart enough to remove it (it concludes each object is still relevant because at least one reference exists). This can happen, for instance, when an object's event handler refers back to the object itself (see "Javascript Closures" [http://jibbering.com/faq/faq_notes/closures.html#clMem] for more details).

Test, test, test.

It may not be fun, but you need to stress test your application under different browser environments, monitoring memory usage to ensure it's stable. (See *System Test* [Chapter 19]).

Real-World Examples

TadaList

TadaList (<http://tadalist.com>; a screencast is available at <http://www.tadalist.com/theater>) allows users to manage TODO items (Figure 5-7). Each TODO item is a phrase, like "Finish homework," and the user can add, remove, and rearrange items.



FIGURE 5-7. TadaList

Super Maryo World

Super Maryo World (<http://www.janis.or.jp/users/segabito/JavaScriptMaryo.html>) is an out-right Ajaxian video game, a clone of the classic Super Mario Bros game implemented with standard Ajax technologies (Figure 5-8). The manipulation of game characters and fixtures illustrates how elements can rapidly be added, removed, and moved around. It's Page Rearrangement, real-time!

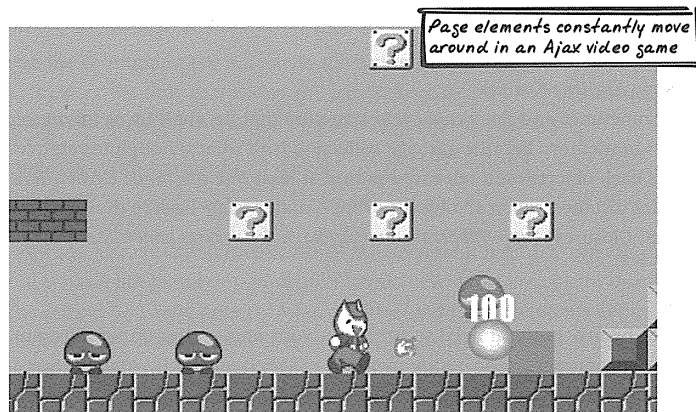


FIGURE 5-8. Super Maryo World

Kiko

Kiko (<http://kiko.com>) is a direct-manipulation Ajax calendar application. You can add and remove appointments, drag them around to change the time, and stretch them out to increase duration.

Code Example: AjaxPatterns Basic Wiki

The Basic Wiki Demo (<http://ajaxify.com/run>) periodically polls for a fresh list of messages to display. Each time the list is retrieved, it removes all existing messages and adds the new list. The containing element is called `messages`, and removing all messages involves running a loop across each of its children:

```
while ($("#messages").hasChildNodes()) {  
    $("#messages").removeChild($("#messages").firstChild);  
}
```

Each message is used to create a new textarea element (among other things), which is then added to a new `div`, which in turn is added to the `messages` container:

```
for (var i=0; i<wikiMessages.length; i++) {  
    var messageArea = document.createElement("textarea");  
    ...  
    messageDiv = document.createElement("div");  
    ...  
    messageDiv.appendChild(messageArea);  
    ...  
    $("#messages").appendChild(messageDiv);  
    ...  
}
```

Related Patterns

Display Morphing

The *Display Morphing* pattern (see earlier in this chapter) addresses an element's appearance; this pattern talks about elements' location, visibility, and "height above the page" (`zIndex`). Together, the two patterns cover all aspects of display manipulation.

Metaphor

Page Rearrangement is like adding, removing, and rearranging Post-It notes on a whiteboard.

Web Remoting

NOW THAT WE'VE SEEN HOW AJAX APPS CAN MANIPULATE THE DISPLAY, IT WOULD BE USEFUL IF THEY could use that capability to show data arriving from the server. We'd also like to accept input and upload it to the server. The Web Remoting patterns let JavaScript directly issue calls to the server. The point is to let the browser make a query or upload some data, without actually refreshing the page.

The first pattern is *Web Service*, which explains how the server side exposes functionality to be accessed by the browser.

The remaining patterns are the most useful and most common mechanisms for web remoting.* *XMLHttpRequest Call* is a clean way for the browser script to invoke and catch responses from the server. *IFrame Call* provides a similar use by exploiting IFrame functionality. *HTTP Streaming*, also known as "Push" (or "Comet"), allows the server to continue streaming new data down the pipe, without the browser having to issue new

* Other techniques do exist, and are mentioned in "Alternatives" in *XMLHttpRequest Call* later in this chapter.

®

requests. *On-Demand JavaScript* is a fairly broad-scoped pattern and is included in this section because one style, Script Tag Generation, is a distinctive technology that's becoming a popular way to directly access external domains.

Of the four techniques, *XMLHttpRequest Call* is the cleanest, because *XMLHttpRequest* is specifically designed for web remoting. Nevertheless, all of the techniques are effective, and each has a certain set of situations where it's the superior choice. See the "Alternatives" section for *XMLHttpRequest Call* for a comparison of the various techniques.

Web Service



API, HTTP, Microcontent, REST, RPC, Share, SOAP, Platform

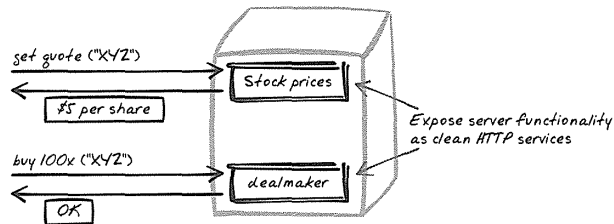


FIGURE 6-1. Web Service

Goal Story

In a financial forecasting app, Tracy runs an algorithm to forecast the next four interest rates. The browser script uploads the parameters to an "Interest Rate Forcaster" web service, which eventually outputs a concise list of future rates, free of any HTML markup: 4.5, 3.4, 4.0, 4.1.

Problem

What will Ajax Apps call on the server?

Forces

- Ajax Apps can access the server directly and require well-scoped services instead of the conventional-style scripts that output an entire HTML page.
- Many organizations like to expose their functionality and data for third parties to access. The API needs to be clean and easy to use.

Solution

Expose server-side functionality as fine-grained, easy-to-use Web Services. "Web Service" is an overloaded term, and the Ajax Patterns use a fairly broad definition:

- A Web Service is a standard HTTP service.
- A Web Service has a well-defined, consistent interface—input and output assumptions, message formats, and exception handling are all clearly understood by its developers and ideally documented in some manner.
- A Web Service accepts fine-grained input parameters and outputs fine-grained responses, such as an XML document, a simple string, or a snippet of HTML. If it outputs an entire HTML document, it's probably not a Web Service.

Under this definition, there is considerable flexibility involved in creating a Web Service:

- A Web Service might be open to the public, might be protected by a secure authentication mechanism, or might be hidden from the public by a firewall.
- A Web Service might be built by a tool or handwritten.
- A Web Service might use messages according to any protocol, such as SOAP, or might use custom messages.

Deciding on all these things is the main task involved in creating a Web Service.

Web services are the latest in a long tradition of distributed computing, with technologies like SunRPC, CORBA, and EJB along the way. They've arisen quite separately from Ajax, as a general-purpose way to expose services to interested clients. These clients are usually standalone processes with no user interface as well as desktop clients and web server scripts. Until recently, they were irrelevant to anything inside a web browser, which could only communicate with services capable of returning an entire HTML page. But thanks to remoting technologies, JavaScript can now make use of such services. Indeed, a clean Web Service is actually the best thing for remoting technologies to call—it makes the call easier, and having a fine-grained response is good for the response handler. Thus, there's a strong synergy between Ajax and web services.

Here's the PHP code for a one-liner web service to perform a sum (imaginative, I know):

```
<?
echo $_GET["figure1"] + $_GET["figure2"];
?>
```

You can try the service by entering *<http://ajaxify.com/run/xmlHttpRequestCall/sumGet.php?figure1=5&figure2=10>* in your browser's address bar. The entire response will be "15." As the example shows, a web service's response can be as simple as a single value. Other times, it might be some HTML, but even then, it will usually only be a snippet rather than an entire page. For example, the above service could be refactored to show the result as a heading:

```
<?
echo "<h1>" . ($_GET["figure1"] + $_GET["figure2"]) . "</h1>";
?>
```

A collection of related services like these forms an HTTP-based API that exposes server functionality. Most Ajax Apps will access the API from the browser via the *XMLHttpRequest* ⁶ *Call* and alternative remoting techniques. In addition, third parties can access Web Services and use them for their own applications, web or not.

This section introduces only the concept of web services. There are a number of decisions you need to make when designing such services for your own application. Will the service output XML to be processed in the browser, HTML to be displayed directly, or some other format? How will the URLs look? What sort of input will it take? All these issues are discussed in Chapter 9.

Decisions

How will the web service be used?

The single most important decision about a web service, as with any form of API, is to know how it will be used. In particular, will it be available for third-parties as a generic interface to your system, or are you developing it purely for an Ajax browser app to access? If the latter, then you might want to practice feature-driven development and let the browser app drive web service design—see *Simulation Service* (Chapter 19).

If there *are* third parties involved, it becomes a balancing act between their needs and the needs of your browser app. Since they will probably be accessing your server from more powerful environments than the average web browser, you might need to add some redundancy to support each type of client. The API for third parties might be more general, whereas the browser sometimes needs an API that knows something about the application. In an extreme case, it tracks application state and outputs the actual HTML to be shown to the user.

How will you prevent third-party usage?

Web services are seen as having a kind of synergy with Ajax, as Ajax Apps can talk to the same interface already being offered to third-party clients. However, sometimes you want the web service to be closed to third parties. That's going to be difficult on the public Web, since pretty much anything a browser can access will also be available to third-party clients. So, you might have spent ages designing a nice, clean web service to make browser scripting as painless as possible, only to discover that third parties are reaping the benefits. There's no magic bullet, but here are a few suggestions:

- Require users to be logged in, with an authenticated email address, in order for the application to make use of web services. You can then use cookies or upload unique session IDs to authenticate the user.
- Consider using a system like Captcha (<http://www.captcha.net/>), where the user is forced to perform a challenge that should be about impossible for an automated script to do. You might require a challenge be solved once an hour, say.

- Use standard filtering techniques, like blocking certain IP addresses and refusing requests if they arrive too frequently.
- Use obfuscation and encryption. This idea is based on a tricky little anti-spam plugin for Wordpress, *HashCash* (<http://wp-plugins.net/plugin/wp-hashcash/>). By outputting a customized piece of obfuscated JavaScript that's used to decrypt web service content, you force any client programmer to do two things: first, hook into a JavaScript engine in order to execute the code; second, consume resources performing the decryption. Neither is impossible, but they do make the task less appealing. Unfortunately, they also break the clean nature of web services, though you should be able to abstract the messy details with a suitable JavaScript API.

Real-World Examples

Just about any Ajax App has some kind of web service. In these examples, we'll look at a public API designed for third-party usage, then at a couple of services accessed only from a corresponding Ajax browser script.

Technorati API

Technorati (<http://technorati.com>) exposes its search capability for third-party use. To perform a query, run an HTTP GET on the following URL: <http://api.technorati.com/search?key=1234&query=serendipity>, where "1234" is your personal technorati key and "serendipity" is the search query. You'll get an XML document that lists the results.

NetVibes

NetVibes (<http://netvibes.com>) is an Ajax portal that gets its content from several web services. To get weather data, the browser invokes a service like this:

```
http://netvibes.com/xmlProxy.php?url=http%3A//xoap.weather.com/weather/local/USNY0996%3Fcc%3D%26unit%3Dd%26dayf%3D4
```

The service is actually a proxy because it passes through to a real weather service at weather.com. As the URL shows, you can specify parameters such as location (USDNYC0996), units (d), and number of days ahead (4). The output is an XML document without any HTML markup:

```
<weather ver="2.0">
  <head>
    <locale>en_US</locale>
    <form>MEDIUM</form>
    ...
  </head>
  <dayf>
    ...
    <hi>N/A</hi>
    <low>46</low>
    <sunr>7:18 AM</sunr>
```

```

<suns>4:34 PM</suns>
...
</dayf>
...
</weather>

```

Wish-O-Matic

Alexander Kirk's Wish-O-Matic (<http://wish-o-matic.com>) provides Amazon recommendations. The browser posts the following information to a service at [http://alexander.kirk.at/wish-o-matic/search=grand&catalog=Books&page=1&locale=US&_ =](http://alexander.kirk.at/wish-o-matic/search=grand&catalog=Books&page=1&locale=US&_=). The service then responds with a list of books. Unlike the previous service, the data is pure HTML, ready for immediate display in the browser.

```

search=grand&catalog=Books&page=1&locale=US&_ =

<table><tr><td style="width: 500px">
  <b>Author:</b> Kevin Roderick<br/><b>Title:</b> Wilshire Boulevard:

  The Grand Concourse of Los Angeles<br/><b>ISBN:</b> 1883318556<br/><a
href="javascript:void(add_item('1883318556', 'Wilshire Boulevard: The Grand
  Concourse of Los Angeles',
'http://images.amazon.com/images/P/1883318556.01._SCTHUMBZZZ_.jpg'))">My
friend likes this item!</a></td>
...
</td></tr>
</table>
<br/>
<a href="javascript:void(prev_page())">&lt; prev</a> |
<a href="javascript:void(next_page())">next &gt;</a>

```

Code Example: AjaxPatterns testAjaxCaller

AjaxCaller is a JavaScript HTTP client used in examples throughout this book (explained later in this chapter in “Code Example: AjaxPatterns TestAjaxCaller” for *XMLHttpRequest Call*). Here, we'll look at a web service used in the testAjaxCaller application (<http://ajaxlocal/run/testAjaxCaller/>), a very simple service that just echoes its input.

httpLogger begins by outputting a request method (e.g., “GET” or “POST”), its own request URL, and CGI variables present in the URL and body:

```

echo "<p>Request Method: $requestMethod</p>";
echo "<p>Request URL: ".$_SERVER['REQUEST_URI']."</p>";
echo "<p>Vars in URL: ".print_r($_GET, TRUE)."</p>";
echo "<p>Vars in Body: ".print_r($_POST, TRUE)."</p>";

```

A separate function is used to read the body, which can also be outputted with a regular print or echo statement:

```
function readBody() {
    $body="";
    $putdata = fopen("php://input", "r");
    while ($block = fread($putdata, 1024)) {
        $body = $body.$block;
    }
    fclose($putdata);
    return $body;
}
```

Related Patterns

Web Services patterns

All the patterns in Web Services (Chapter 9) explain various strategies for designing Web Services with clean, maintainable interfaces.

Web Remoting patterns

The remaining Web Remoting patterns explain how the browser invokes a server-side Web Service.

Cross-Domain Proxy

The main point of this pattern is to guide on designing your own Web Services pattern. However, there are times when your server script will need to call on an external web service, as explained in *Cross-Domain Proxy* (Chapter 10).

Simulation Service

A *Simulation Service* (Chapter 19) is a “dummy” service that produces canned responses, a useful device while developing the browser-side of an Ajax App.

Service Test

The nice thing about web services, compared to most aspects of web development, is that it’s easy to write automated tests, as described in *Service Test* (Chapter 19).

XMLHttpRequest Call



Call, Callback, Download, Grab, Live, Query, Remoting, RemoteScripting, Synchronise, Synchronize, Upload, XMLHttpRequest

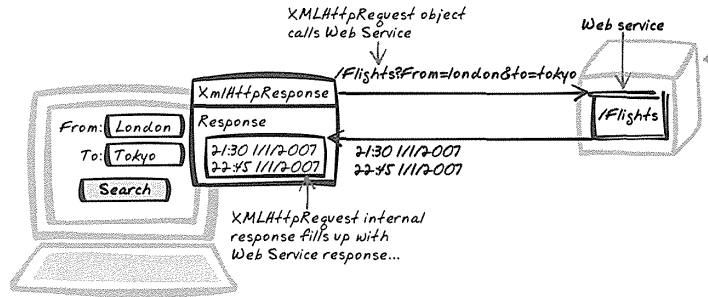


FIGURE 6-2. XMLHttpRequest Call

Goal Story

Reta's purchasing some items from a wholesaler's web site. Each time she adds an item to the shopping cart, the web site issues an XMLHttpRequest to save the latest cart contents. There's no form submission, so the item is added instantaneously, which saves Reta time as well as helping her understand what's going on.

Problem

How can the browser communicate with the server?

Forces

- Ajax Apps require browser-server communication. User-generated information must be uploaded and new server information must be downloaded.
- Because Ajax Apps should have a smooth and continuous feel, browser-server communication must be unobtrusive.
- Ajax Apps should be highly responsive, so calls should involve minimal data transfer.
- As the network is often unreliable and performance is inconsistent, calls should be asynchronous, allowing the user to keep working while network calls are in progress.

Solution

Use XMLHttpRequest objects for browser-server communication. JavaScript lacks a portable mechanism for general network communication, a restriction that's always been in place for security reasons and will probably remain. But thanks to the XMLHttpRequest object—now available in all major browsers—JavaScript code *can* make HTTP calls back to its originating server and get hold of the results. Doing so enables you to make fine-grained server calls and deal with responses as you wish, unlike conventional form submissions, which cause a complete page refresh. Note: an online demo (<http://ajaxify.com/run/xmlHttpRequestCall>) illustrates the code concepts throughout this Solution and the code snippets loosely follow from the demo.

This pattern uses the sum web service described earlier in *Web Service*, with a URL like `http://ajaxify.com/run/xmlHttpRequestCall/sumGet.phtml?figure1=5&figure2=10`. It returns just the sum, “15” in this case. You can test that by typing the full URL in the browser, but here we want to call it from JavaScript and catch the result. Here’s a very basic example:

```
var xhrReq = new XMLHttpRequest();
xhrReq.open("GET", "sumGet.phtml?figure1=5&figure2=10", false);
xhrReq.send(null);
var serverResponse = xhrReq.responseText;
alert(serverResponse); // Shows "15"
```

The sequence begins by creating a new instance of `XMLHttpRequest`. `xhrReq.open()` then prepares a call on the test service, `sumGet.phtml` (the code’s running from the same path, so the domain and path need not be qualified). The `GET` signifies the request method to be used. The `false` argument says the call is synchronous, meaning that the code will block until a response comes back. The `send` command completes the request. Because the call is synchronous, the result is ready as soon as the next line is executed. The `XMLHttpRequest` object has saved the response from the server, and you can access it with the `responseText` field.

The above example shows that the fundamental technology is pretty simple. However, be aware that it’s a very basic usage that’s not yet fit for production. Fundamental questions remain, which are answered throughout this Solution:

- How do you get hold of an `XMLHttpRequest`?
- How do *asynchronous* calls work?
- How do you handle errors?
- What if the service requires a `POST` or `PUT` request rather than a `GET`?
- What constraints apply to external domains?
- How can you deal with XML responses?
- What’s the API?

As you read all this, be aware that many, many libraries are available to handle remoting (see Appendix A). **Most developers shouldn’t need to touch `XMLHttpRequest` directly.** Having said that, it’s good to be aware of the capabilities and limitations of `XMLHttpRequest` Calls, along with the other web-remoting techniques. This knowledge will help you select the most appropriate library and help with any bugs you might encounter.

Creating `XMLHttpRequest` objects

In most browsers, `XMLHttpRequest` is a standard JavaScript class, so you just create a new instance of `XMLHttpRequest`. However, Microsoft is the inventor of `XMLHttpRequest`, and until IE7, IE offered it only as an ActiveX object. To make things even more fun, there are different versions of that object. The following code shows a factory function that works on all browsers that support `XMLHttpRequest`:

```
function createXMLHttpRequest() {
    try { return new XMLHttpRequest(); } catch (e) {}
}
```

```

    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}
    try { return new XMLHttpRequest(); } catch(e) {}
    alert("XMLHttpRequest not supported");
    return null;
}
...
var xhrReq = createXMLHttpRequest();

```

You really need to use a function like this for maximum portability. Once you have the object, its basic functionality and API are pretty consistent across browsers, but be sure to test carefully as there are a few subtle implementation differences in some browsers. (If you're curious, the "Solution" in *HTTP Streaming* [later in this chapter] highlights one such inconsistency.)

You can also reuse an XMLHttpRequest; it's worthwhile doing so in order to prevent memory leaks. To be safe, start a new call only when there's not one already in progress. As explained below, it's possible to inspect the status of a call, and you should only start a call if the status is 0 or 4. So if it's anything else, first call the `abort()` method to reset status.

Asynchronous calls

I previously mentioned in the "Solution" that under synchronous mode, "the code will block until a response comes back." Some hardened readers probably writhed uncomfortably at the thought. We all know that some requests take a long time to process, and some don't come back at all. Pity the user when a server script is buried in an infinite loop.

In practice, XMLHttpRequest Calls should almost always be asynchronous. That means the browser and the user can continue working on other things while waiting for a response to come back. How will you know when the response is ready? The XMLHttpRequest's `readyState` always reflects the current point in the call's lifecycle. When the object is born, it's at 0. After `open()` has been called, it's 1. The progression continues until the response is back, at which point the value is 4.

So, to catch the response, you need to watch for a `readyState` of 4. That's easy enough, because XMLHttpRequest fires `readystatechange` events. You can declare a callback function using the `onreadystatechange` field. The callback will then receive all state changes. The states below 4 aren't especially useful and are somewhat inconsistent across browser types anyway (http://www.quirksmode.org/blog/archives/2005/09/xmlhttp_notes_r_2.html). So most of the time, all we're interested in is, "Are you in state 4 (i.e., complete) or not?"

Based on all that, here's an asynchronous version of the code shown earlier:

```

var xhrReq = createXMLHttpRequest();
xhrReq.open("GET", "sumGet.phtml?figure1=5&figure2=10", true);
xhrReq.onreadystatechange = onSumResponse;
xhrReq.send(null);
...
function onSumResponse() {
    if (xhrReq.readyState != 4) { return; }
    var serverResponse = xhrReq.responseText;
    ...
}

```


As shown, you declare the callback method in XMLHttpRequest's `onreadystatechange` property. In addition, the third argument of `open()` is now `true`. This argument is actually called the "asynchronous flag," which explains why we're now setting it to `true`. The callback function, `onSumResponse`, is registered using `onreadystatechange` and contains a guard clause to ensure the `readyState` is 4 before any processing can occur. At that point, we have the full response in `responseText`.

JavaScript also supports "closures"—a form of anonymous function—which suggests a more concise boilerplate structure for asynchronous calls:

```
var xhrReq = createXMLHttpRequest();
xhrReq.open("get", "sumget.php?figure1=10&figure2=20", true);
xhrReq.onreadystatechange = function() {
    if (xhrReq.readyState != 4) { return; }
    var serverResponse = xhrReq.responseText;
    ...
};
xhrReq.send(null);
```

Use closures sparingly, because you're defining a new function each time. It's slower than referring to an existing one and might also lead to memory leaks.

Asynchronous calls are essential, but also more error-prone. If you look at the callback mechanism, you might notice the potential for a subtle, but serious, bug. The problem arises when the same instance of XMLHttpRequest is simultaneously used for different calls. If Call 2 is issued while the object is still waiting for the response of Call 1, what will the callback function receive? In fact, it's even possible the callback function itself is changed before the first call returns. There are ways to deal with this problem, and they're the topic of the *Call Tracking* (Chapter 10) pattern.

Detecting errors

Sometimes, a request doesn't come back as you expected it, or maybe not at all. You scripted the call wrong, or there's a bug in the server, or some part of the infrastructure just screwed up. Thinking asynchronously is the first step to dealing with these problems, because at least your application isn't blocked. But you need to do more than that.

To detect a server error, you can check the response status using XMLHttpRequest's `status` flag. This is just a standard HTTP code. For example, if the resource is missing, XMLHttpRequest.`status` will take on the famous "404" value. In most cases, you can assume anything other than 200 is an error situation. This suggests adding a new check to the callback function of the previous section:

```
xhrReq.onreadystatechange = function() {
    if (xhrReq.readyState != 4) { return; }
    if (xhrReq.status != 200) {
        var serverResponse = xhrReq.responseText;
        ...
    }
};
```

That's great if the browser knows a problem occurred, but sometimes the request will be lost forever. Thus, you usually want some kind of timeout mechanism (<http://ajaxblog.com/archives/2005/06/01/async-requests-over-an-unreliable-network>) as well. Establish a *Scheduling* timer to track the session. If the request takes too long, the timer will kick in and you can then handle the error. XMLHttpRequest has an `abort()` function that you should also invoke in a timeout situation. Here's a code sample:

```
var xhrReq = createXMLHttpRequest();
xhrReq.open("get", "infiniteLoop.phtml", true); // Server stuck in a loop.
var requestTimer = setTimeout(function() {
    xhrReq.onreadystatechange = function() {
        if (xhrReq.readyState != 4) { return; }
        clearTimeout(requestTimer);
        if (xhrReq.status != 200) {
            // Handle error, e.g. Display error message on page
            return;
        }
        var serverResponse = xhrReq.responseText;
        ...
    };
};
```

Compared to the previous example, a timer has been introduced. The `onreadystatechange()` callback function will clear the timer once it receives the full response (even if that response happens to be erroneous). In the absence of this clearance, the timer will fire, and in this case, the `setTimeout` sequence stipulates that `abort()` will be called and some recovery action can then take place.

Handling POSTs and other request types

Up to this point, requests have been simple GET queries—pass in a URL and grab the response. As discussed in the *RESTful Service* (Chapter 9), real-world projects need to work with other request types as well. POST, for example, is suited to calls that affect server state or upload substantial quantities of data. To illustrate, let's now create a new service, `sumPostGeneric.phtml`, that does the same thing as `postGet.phtml` but with a POST message. It's called "generic" because it reads the full message body text, as opposed to a CGI-style form submission. In this case, it expects a body such as "Calculate this sum: 5+6" and returns the sum value:

```
<?
$body = readBody();
ereg("Calculate this sum: ([0-9]+)\+([0-9]+)", $body, $groups);
echo $groups[1] + $groups[2];

// A PHP method to read arbitrary POST body content.
function readBody() {
    $body="";
    $putData = fopen("php://input", "r");
    while ($block = fread($putData, 1024)) {
        $body = $body.$block;
    }
    fclose($putData);
}
```

```

        return $body;
    }
    ?>

```

To POST an arbitrary body, we give `XMLHttpRequest` a request type of `POST` and pass the body in as an argument to `send()`. (Note that with `GET` queries, the `send()` argument is null as there's no body content).

```

var xhr = new XMLHttpRequest();
xhr.open("post", "sumPostGeneric.phtml", true);
xhr.onreadystatechange = function() {
    if (xhr.readyState != 4) { return; }
    var serverResponse = xhr.responseText;
    ...
};
xhr.send("calculate this sum: 5+6");

```

Quite often, though, you'll be posting key-value pairs, so you want the message to look as if it were submitted from a `POST`-based form. You'd do that because it's more standard, and server-side libraries make it easy to write web services that accept standard form data. The service shown in the code example following the next one, `sumPostForm.php`, shows how PHP makes light work of such submissions, and the same is true for most languages:

```

<?
    echo $_POST["figure1"] + $_POST["figure2"];
?>

```

For the browser script to make a `CGI`-style upload, two additional steps are required. First, declare the style in a `"Content-Type"` header; as the example below shows, `XMLHttpRequest` lets you directly set request headers. The second step is to make the body a set of name-value pairs:

```

var xhr = new XMLHttpRequest();
xhr.open("post", "sumPostForm.phtml", true);
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
xhr.onreadystatechange = function() {
    if (xhr.readyState != 4) { return; }
    var serverResponse = xhr.responseText;
    ...
};
xhr.send("calculate this sum: 5+6");

```

`GET` and `POST` are virtually ubiquitous, but *RESTful Service* points out there's a time and place for other request methods too, such as `PUT` and `DELETE`. You don't have to do anything special with those other methods; just set the request type in the `open()` call and `send()` an appropriate body (the item you're putting in the case of `PUT`; a null argument in the case of `DELETE`).

Constraints on external domains

On discovering `XMLHttpRequest`, a common reaction is to start dreaming up an interface that pulls in content from popular web sites and mashes it altogether to into one big Web 2.0

soufflé. Unfortunately, it's not so simple because of a key security rule imposed by all major browsers: XMLHttpRequest can only access content from the originating server. If your application lives at <http://ajax.shop/admin>, then your XMLHttpRequest objects can happily reach <http://ajax.shop/admin/products.html> and <http://ajax.shop/products/contents.html>, shouldn't be able to reach <http://books.ajax.shop/contents.html>, and definitely won't have access to <http://google.com>.

This "same-origin policy" (or "same-domain policy") (<http://www.mozilla.org/projects/security/components/same-origin.html>) will be familiar to developers of Java applets and Flash, where the policy has always been in place. It's there to prevent all kinds of abuse, such as a malicious script grabbing confidential content from one server and uploading it to another server under their own control. Some have suggested it's possibly overkill, that most of the risks it tries to prevent are already possible by other means (<http://spaces.msn.com/members/siteexperts/Blog/cns!1pNcL8JwTfkkjv4gg6LkVCpw!2085.entry>). However, restrictions like this won't be lifted lightly; the rule's likely to be around for the long term, so we had better learn to work with it.

Given same-origin restrictions, then, how do all those Ajax mashup sites work (<http://housingmaps.com>)? The answer is that the cross-domain transfers usually run through the originating server, which acts as a kind of proxy—or tunnel—allowing XMLHttpRequests to communicate with external domains. *Cross-Domain Proxy* (Chapter 10) elaborates on the pattern, and its "Alternatives" section lists some clever workarounds that do allow the originating server to be bypassed.

XML responses

The discussion here has swiftly ignored the big elephant in the room: XML. XMLHttpRequest, as its name suggests, was originally designed with, yes, XML in mind. As we've already seen, it will actually accept any kind of response, so what's special about XML? With XMLHttpRequest, any responses can be read via the `responseText` field, but there's also an alternative accessor: `responseXML`. If the response header indicates the content is XML, and the response text is a valid XML string, then `responseXML` will be the DOM object that results from parsing the XML.

The Display Manipulation (Chapter 5) patterns have already illustrated how JavaScript supports manipulation of DOM objects. In those patterns, we were only interested in one particular DOM object, the HTML (or XHTML) document representing the current web page. But you can manipulate any other DOM object just as easily. Thus, it's sometimes convenient to have a web service output XML content and manipulate the corresponding DOM object.

The prerequisite here is a *Web Service* (see earlier in this chapter) that outputs valid XML. There are many libraries and frameworks around for automatically generating XML from databases, code objects, files, or elsewhere. But don't think you have to start learning some fancy XML library in order to create XML web services, because it's fairly easy to hand code them too, at least for simple data. The service just needs to output an XML

Content-type header followed by the entire XML document. Here's an XML version of the sum service shown earlier—it outputs an XML document containing the input figures as well as the sum result:

```
<?
  header("Content-Type: text/xml");
  $sum = $_GET["figure1"] + $_GET["figure2"];
  echo <<< END_OF_FILE
<sum>
  <inputs>
    <figure id="1">{$_GET["figure1"]}</figure>
    <figure id="2">{$_GET["figure2"]}</figure>
  </inputs>
  <outputs>$sum</outputs>
</sum>
END_OF_FILE
?>
```

The call sequence is the same as before, but the callback function now extracts the result using `responseXML`. It then has a first-class DOM object and can interrogate it using the standard DOM API:

```
var xhrReq = createXMLHttpRequest();
xhrReq.open("GET", "sumXML.php?figure1=10&figure2=20", true);
xhrReq.onreadystatechange = function() {
  if (xhrReq.readyState != 4) { return; }
  xml = xhrReq.responseXML;
  var figure1 = xml.getElementsByTagName("figure")[0].firstChild.nodeValue;
  var figure2 = xml.getElementsByTagName("figure")[1].firstChild.nodeValue;
  var sum = xml.getElementsByTagName("outputs")[0].firstChild.nodeValue;
  ...
};
xhrReq.send(null);
});
```

The name “XMLHttpRequest” relates to its two primary functions: handling HTTP requests and converting XML responses. The former function is critical and the latter is best considered a bonus. There are certainly good applications for XML responses—see *XML Message* (Chapter 9), and *XML Data Island* and *Browser-Side XSLT* (Chapter 11)—but keep in mind that XML is not a requirement of Ajax systems.

You can also upload XML from browser to server. In this case, `XMLHttpRequest` doesn't offer any special XML functionality; you just send the XML message as you would any other message, and with an appropriate request type (e.g., POST or PUT). To support the receiving web service, the JavaScript should generally declare the XML content type in a request header:

```
xhrReq.setRequestHeader('Content-Type', "text/xml");
```

The XMLHttpRequest API: a summary

We've looked at how to achieve typical tasks with `XMLHttpRequest`, and now here's a quick summary of its properties and methods based on an Apple Developer Connection article

(<http://developer.apple.com/internet/webcontent/xmlhttpreq.html>). The API is supported by IE5+, the Mozilla family (including all Firefox releases), and Safari 1.2+.

XMLHttpRequest has the following properties:

`onreadystatechange`

The callback function that's notified of state changes. 0=UNINITIALIZED, 1=LOADING, 2=LOADED, 3=INTERACTIVE, and 4=COMPLETE. (As explained earlier in "Asynchronous calls," states 1–3 are ambiguous and interpretations vary across browsers.)

`readyState`

The state within the request cycle.

`responseText`

The response from the server, as a String.

`responseXML`

The response from the server, as a Document Object Model, provided that the response "Content-Type" header is "text/html," and the `responseText` is a valid XML string.

`status`

HTTP response code (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>) received from the server. This should normally be 200; most values indicate an error.

`statusText`

The HTTP response code description received from the server; e.g., "Not Found."

And these are XMLHttpRequest's methods:

`abort()`

Stops the request and resets its `readyState` back to zero. (See "Detecting errors," earlier in this chapter.)

`getAllResponseHeaders()`

Returns a string of all response headers, separated by a newline as in the original message.

`getResponseHeader(headerField)`

Returns the value for a particular header field.

`open(requestMethod, url, asynchronousFlag, username, password)`

Prepares XMLHttpRequest (See the "Solution," earlier.). Only the first two parameters are optional. `username` and `password` can be used for authentication.

`send(bodyContent)`

Sends the message along with specified body content (null if no body content is to be sent; e.g., for GET requests). (See the "Solution," earlier.)

`setRequestHeader(headerField, headerValue)`

Sets a request header. (See "Handling POSTs and other request types," earlier.)

Decisions

What kind of content will web services provide?

As mentioned in the solution, XML is not the only kind of content that XMLHttpRequest can deal with. As long as you can parse the message in JavaScript, there are various response types possible. The patterns on web services highlight a number of response types, including HTML, XML, JSON, and plain-text.

How will caching be controlled?

It's possible that an XMLHttpRequest response will be cached by the browser. Sometimes, that's what you want and sometimes it's not, so you need to exert some control over caching.

With cache control, we're talking about GET-based requests. Use GET for read-only queries and other request types for operations that affect server state. If you use POST to get information, that information usually won't be cached. Likewise, if you use GET to *change* state, you run the risk that the call won't always reach the server, because the browser will cache the call locally. There are other reasons to follow this advice too; see *RESTful Service* (Chapter 9).

Often, you want to suppress caching in order to get the latest server information, in which case, a few techniques are relevant. Since browsers and servers vary, the standard advice is spread the net as wide as possible by combining some of these techniques:

- You can make the URL unique by appending a timestamp (<http://www.howtoadvice.com/StopCaching>) (a random string, or a string from an incrementing sequence, is sometimes used too). It's a cheap trick, but surprisingly robust and portable:

```
var url = "sum.phtml?figure1=5&figure2=1&timestamp=" + new Date().getTime();
```

- You can add a header to the request:

```
xhReq.setRequestHeader("If-Modified-Since", "Sat, 1 Jan 2005 00:00:00 GMT");
```

- In the *Web Service* set response headers to suppress caching (<http://www.stridebird.com/articles/?showarticle=1&id=33>). In PHP, for example:

```
header("Expires: Sat, 1 Jan 2005 00:00:00 GMT");  
header("Last-Modified: ".gmdate("D, d M Y H:i:s")."GMT");  
header("Cache-Control: no-cache, must-revalidate");  
header("Pragma: no-cache");
```

- Use POST instead of GET. Requests that are of POST type will sometimes cause caching to be suppressed. However, this particular technique is not recommended because, as explained in *RESTful Service*, GET and POST have particular connotations and shouldn't be treated as interchangeable. In any event, it won't always work, because it's possible some resources will actually cache POST responses.

On the other hand, caching is a good thing when the service is time-consuming and unlikely to have changed recently. To encourage caching, you can reverse the above

advice; e.g., set the Expires headers to a suitable time in the future. In addition, a good approach for smaller data is to cache it in the program itself, using a JavaScript data structure. *Browser-Side Cache* (Chapter 13) explains how.

How will you deal with errors?

The section on error detection left open the question of what to do once we discover a server timeout or nonstandard error code. There are three possible actions:

Try again

Retry a few times before giving up.

Inform the user

Tell the user what's gone wrong and what the consequences are. For instance, inform him that his data hasn't been submitted and he should try again in a few minutes.

Do nothing

Sometimes, you have the luxury of ignoring the response (or lack thereof). That might be because you're issuing low-importance Fire-and-Forget calls (http://www.ajaxian.com/archives/2005/09/ajaxian_fire_an.html), where you're uploading some data without waiting for any response.

Real-World Examples

Lace Chat

Brett Stimmerman's Lace Chat (<http://www.socket7.net/lace/>) is an Ajax chat application that uses XMLHttpRequest in two ways: to upload messages you type and to download all the latest messages from the server (Figure 6-3).

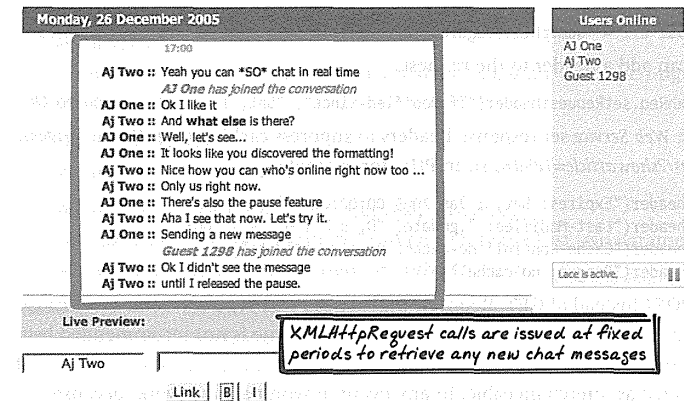


FIGURE 6-3. Lace Chat

Backbase

Backbase's Demo RSS Reader (<http://www.backbase.com/demos/RSS>) uses XMLHttpRequest to pull down titles of recent articles (Figure 6-4). When you click on one of those titles, a new XMLHttpRequest will pull down the entire content.

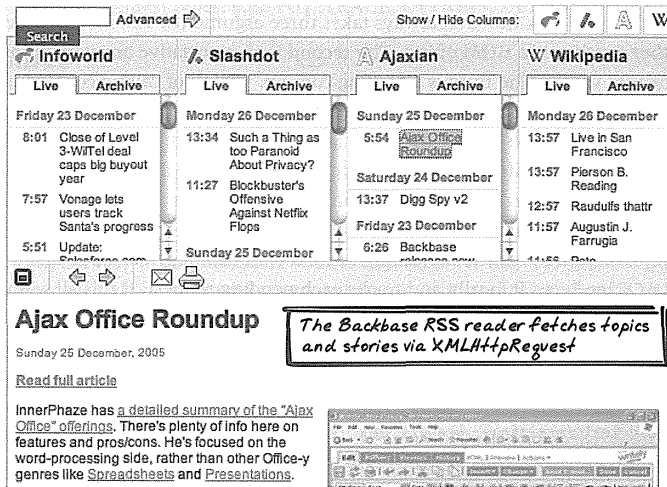


FIGURE 6-4. Backbase RSS Reader

Anyterm

Phil Endecott's Anyterm (<http://anyterm.org/demos.html>) is an Ajax terminal emulator allowing you to run telnet or SSH within the browser. It uses XMLHttpRequest Calls to upload keystrokes and download the latest screen state.

Mint

Mint (<http://haveamint.com/>) is a web site statistics package. Site owners include Mint JavaScript on each page, which quietly inspects the user's browser settings and uploads them using an XMLHttpRequest.

Code Example: AjaxPatterns TestAjaxCaller

The example (<http://ajaxpatterns.org/xmlHttpRequestCall/>) referenced in the "Solution," earlier, covers most typical XMLHttpRequest usage. In practice, many people adopt Ajax frameworks and libraries rather than calling XMLHttpRequest directly. That's the approach taken with all of the Ajax Patterns demos, which use a library called *ajaxCaller.js* that was developed in parallel to the demos themselves. It's a fairly basic library, but offers a simple interface for the functionality that's typically required of XMLHttpRequest. In this section, I'll

introduce the library by showing a few usages within the AjaxCaller Test Demo (<http://ajaxify.com/run/testAjaxCaller>).

The simplest call is getting some plain-text: just specify the URL and the callback function.

```
ajaxCaller.getPlainText(url, onResponse);
```

For all calls, the callback function always takes three arguments. The first argument is the result, either a string or a DOM object. The second is an associative array mapping header fields to header values. The third is a “calling context.” Think of calling context as an optional value that travels alongside the request and the corresponding response, returned to the callback function in exactly the same form as you passed it in when the call was issued. Usually it holds information about the call; e.g., if the call was made to send off a purchase order, the calling context might contain the item that was ordered. Then, `ajaxCaller` will pass the context into the callback function, which can mark the item as successfully ordered. In reality, the calling context is not actually passed to and from the server; `ajaxCaller` keeps it locally and tracks each pending request. If this all sounds a bit complicated, check out *Call Tracking* (Chapter 10).

The callback function looks as follows:

```
function onResponse(text, headers, callingContext) {  
    // Use text (a string), headers, and callingContext  
}
```

And since it’s only the text that’s used most of the time, the function can also be declared in a simpler form.*

```
function onResponse(text) {  
    // Use text (a String)  
}
```

`getPlainText()` is one of four commonly used methods. The others are `getXML()`, `postForPlainText()`, and `postForXML()`. Together, these four cover both common request types (GET and POST) and both response types (text and XML).

```
ajaxCaller.getXML(url, callbackFunction);  
ajaxCaller.postForXML(url, vars, callbackFunction);  
ajaxCaller.getPlainText(url, callbackFunction, callingContext);  
ajaxCaller.postForPlainText(url, callbackFunction, callingContext);
```

There are also a number of more general methods—for example, `get()` provides more flexible GET requests. In addition to a URL and a callback function, `get()` lets you specify some variables to be appended to the URL, a flag to indicate whether the response is XML, and the `callingContext` as discussed above.

```
var vars = {  
    flavour: "chocolate",
```

* Because of the way JavaScript handles function calls, the library call will still go to the function in this form, even though the call contains three arguments.

```

        topping: "nuts"
    };
    ajaxCaller.get("httpLogger.php", vars, onResponse, false, "iceCreamRequest");

```

There are general operations for other request types too. `postVars()` creates a CGI-style POST upload and `postBody()` creates an arbitrary-body POST upload. There are similar methods for other request types; e.g., PUT, TRACE, OPTIONS, DELETE, and HEAD.

Alternatives

This section lists all alternatives I'm aware of, some more limited than others. The more obscure techniques are included for the sake of completeness and also in the hope they might spark a few ideas.

Page refreshes

The conventional way to communicate with the server is for the browser to request an entirely new page, which is pretty extreme when you stop and think about it. It might be appropriate if the user's navigating to a completely different part of a web site, but it's overkill if you want to update a football score at the bottom of the page or upload some user input. The most familiar kind of full page refresh is the hyperlink, which causes the browser to issue a GET request, clear the current page, and output the response. The other kind of full page refresh is a form submission, which causes the browser to pass some parameters with the request—which will be GET, POST, or some other method—and, as with a hyperlink, replace the previous page with the new response. With web remoting, any user-interface changes are completely at the discretion of the script running inside the page. These conventional techniques are still available, but most server communication uses XMLHttpRequest Call and related technologies.

IFrame Call

IFrame Call (see later in this chapter) is the main alternative to XMLHttpRequest. Like XMLHttpRequest, it allows for remote calls using GET, POST, and other request types. But whereas XMLHttpRequest is designed specifically for web remoting, IFrame Call exploits the IFrame to do something it was never really intended to do, and the code shows it. Here's a summary of XMLHttpRequest's strengths over IFrame Calls:

- Being designed specifically for web remoting, the XMLHttpRequest API is easier to use, especially when it comes to non-GET request types. However, this is no great advantage as it's generally recommended that you use a wrapper library to avoid working with either API. (XMLHttpRequest's API may be better, but it's not great!)
- XMLHttpRequest offers functionality not available to *IFrame Call*, such as the ability to abort a call and track the call's state. This can have important performance implications (http://www.ajaxian.com/archives/2005/09/ajaxian_fire_an.html).
- XMLHttpRequest is typically faster, especially with shorter responses (<http://me.eae.net/archive/2005/04/02/xml-http-performance-and-caching/>).
- XMLHttpRequest parses XML in a simple, portable, manner; IFrame is unrelated to XML.

- On those browsers that do support XMLHttpRequest, the API is more consistent than that of IFrame.
- XMLHttpRequest is rapidly gaining the virtue of widespread familiarity. This not only helps other developers understand your code, but also means you benefit from tools such as those which monitor XMLHttpRequest traffic (see *Traffic Sniffing* [Chapter 18]).

For all these reasons, XMLHttpRequest should be the default choice. However, there are some specialized situations where *IFrame Call* is superior:

- IFrame works on many older browsers that don't actually support XMLHttpRequest.
- IFrame happens to have some specialized properties (if only by complete fluke) for browser history and bookmarkability, at least for IE, as discussed in *Unique URLs* (Chapter 17).
- XMLHttpRequest on IE won't work if security measures disable ActiveX, a policy sometimes enforced in the enterprise (<http://verens.com/archives/2005/08/12/ajax-in-ie-without-activex/>).
- IFrame may offer a more portable solution for *HTTP Streaming* as discussed in that pattern.

HTTP Streaming

HTTP Streaming (see later in this chapter) also allows for web remoting, and unlike XMLHttpRequest, the connection remains open. Functionally, the key advantage over XMLHttpRequest is that the server can continuously push new information to the browser. From a resource perspective, streaming is good insofar as there's less starting and stopping of connections, but there are serious scalability issues as it's rarely feasible to keep open a huge amounts of connections and maintain numerous server-side scripts.

Richer Plugin

The *Richer Plugin* (Chapter 8) pattern discusses Java, Flash, and other plugins and extensions. These components often have permission to call the server programmatically, and in some cases, can be used as proxies available to JavaScript code.

On-Demand JavaScript

On-Demand JavaScript (see later in this chapter) describes a couple of ways to download JavaScript on the fly. One involves XMLHttpRequests (and therefore isn't foundational in itself), but the other is an alternative transport mechanism, a different technique for web remoting. It works by adding a script element to the document body, which has the effect of automatically pulling down a named JavaScript file.

Image-Cookie Call

Brent Ashley's RSLite library (<http://www.ashleyit.com/rs/rslite/>) is an unusual alternative based on images and cookies. An image's source property is set to the service URL, which is simply a means of invoking the service. The service writes its response into one or more cookies, which will then be accessible from the JavaScript once the call has completed.

Stylesheet Call

Another way to get at server state is to dynamically change a CSS stylesheet. Just like setting a new JavaScript or image source, you set a stylesheet's href property to point to the web service. In Julien Lamarre's demo of this technique (http://zingzoom.com/ajax/ajax_with_stylesheet.php), the web service actually outputs a stylesheet, and the response is embedded in the URL of a background-image property!

204 Response

An old—and pretty much obsolete—trick is to have the server respond with a 204 “No Content” response code. Browsers won't refresh the page when they see this code, meaning that your script can quietly submit a form to such a service with no impact on the page. However, you can only use the 204 trick for “fire-and-forget” calls—while the response may contain information embedded in the headers, there's no way for a browser script to access it.

Import XML Document

There's a technique specifically for retrieving XML documents that uses similar technology to XMLHttpRequest. Peter-Paul Koch described the technique back in 2000 (<http://www.quirksmode.org/dom/importxml.html>), and recently suggested it can now be written off (http://www.quirksmode.org/blog/archives/2005/01/with_httpmapsea.html).

Metaphor

An XMLHttpRequest Call is like the browser having a side conversation with the server while carrying on the main conversation with the user.

IFrame Call



Call, Callback, Download, Frame, IFrame, Live, Query, Remoting, RemoteScripting, Upload

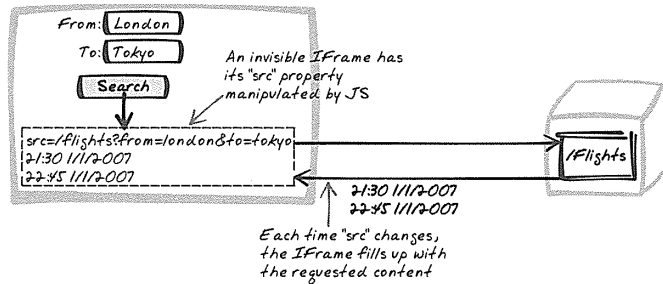


FIGURE 6-5. IFrame Call

Goal Story

Bill's ordering a car online and the price is always synchronized with his choice. This happens because of a hidden IFrame. Each time he changes something, the browser populates a form in the IFrame and submits it. The IFrame soon contains the new price, which the browser copies into the visible display. XMLHttpRequest could have accomplished the feat too, but it's not supported in Bill's outdated browser.

Problem

How can the browser communicate with the server?

Forces

Refer to the section "Forces" in *XMLHttpRequest Call*.

Solution

Use IFrames for browser-server communication. Much of the fuss in Ajax concerns the XMLHttpRequest object, but sometimes other remoting techniques are more appropriate, and IFrame Call is one of those (see the "Alternatives" section of *XMLHttpRequest Call*, earlier in this chapter, for a comparison). IFrames are page-like elements that can be embedded in other pages. They have their own source URL, distinct from their parent page, and the source URL can change dynamically. IFrame Calls work by making the IFrame point to a URL we're interested in, and then reading the IFrame's contents once the new content has loaded. Note: an online demo (<http://ajaxify.com/run/iframe>) illustrates most of the code concepts throughout this Solution and the code snippets loosely follow from the demo.

To begin with, you need an IFrame in the initial HTML, and an event handler registered to catch the onload event. (It would be less obtrusive to register the handler with JavaScript, but due to certain browser “features,” that doesn’t always work.)

```
<iframe id='iFrame' onload='onIFrameLoad();'></iframe>
```

We’re using the IFrame as a data repository, not as a user interface, so some CSS is used to hide it. (The more natural choice would be `display: none`, but as an experimental demo shows [<http://ajaxlocal/run/iframe/displayNone/>], browser strangeness makes it infeasible.)

```
#iFrame {
  visibility: hidden;
  height: 1px;
}
```

On the server, we’re initially going to use the web service discussed in *XMLHttpRequest Call*: `sumGet.phtml`. To call the service, we’ll change the IFrame’s source:

```
$("#iFrame").src = "sumGet.phtml?figure1=5&figure2=1";
```

We already registered an onload handler in the initial HTML. The handler will be called when the new content has loaded, and at that stage, the IFrame’s body will reflect the content we’re after. The following code calls a function, `extractIFrameBody()`, which extracts the body in a portable manner (based on <http://developer.apple.com/internet/webcontent/iframe.html>).

```
function onIFrameLoad() {
  var serverResponse = extractIFrameBody($("#iFrame")).innerHTML;
  $("#response").innerHTML = serverResponse;
}
```

That covers the GET call. To illustrate POSTing, we’ll invoke the `sumPost.phtml` service, also discussed in *XMLHttpRequest Call*. POST calls are a bit more complicated because we can’t just change the source. Instead, we dynamically inject a form into the IFrame and submit it. In this example, the arguments ('5' and '2') are hardcoded, but they could easily be substituted by manipulating the input elements.

```
var iFrameBody = extractIFrameBody($("#iFrame"));
iFrameBody.innerHTML =
  "<form action='sumPostForm.phtml' method='POST'>" +
  "<input type='text' name='figure1' value='5'>" +
  "<input type='text' name='figure2' value='2'>" +
  "</form>";
var form = iFrameBody.firstChild;
form.submit();
```

As with conventional form submission, the IFrame will be refreshed with the output of the form recipient. Thus, the previous onload handler will work fine. Figure 6-6 illustrates the overall process.

Another approach with IFrames is to have the server output some JavaScript that alters the parent or calls a function defined in the parent. The code will automatically be called when the page loads, so there’s no need for an onload handler. Unfortunately, this

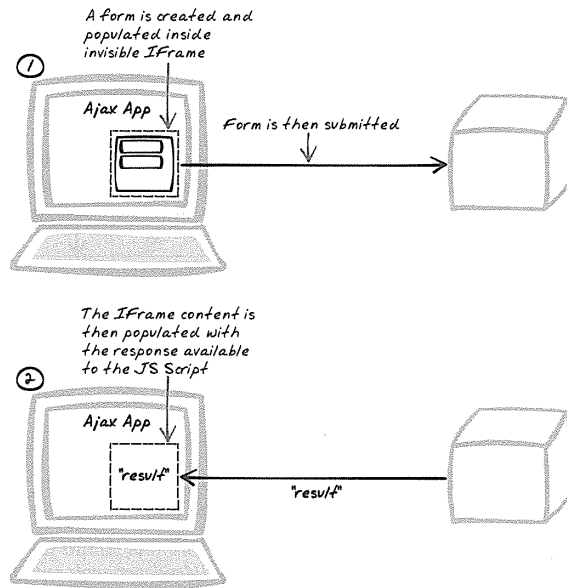


FIGURE 6-6. Posting a form via IFrame Call

approach creates substantial coupling between the server and browser. In contrast, note how the examples above used completely generic web services—the services only know how to add numbers together and nothing about the calling context. That’s possible because all behavior was encapsulated in the `onload()` function.

As *Call Tracking* (Chapter 10) discusses, there are problems—and suitable workarounds—with simultaneous *XMLHttpRequest* Calls, and the same is true for IFrames. First, you need to create a separate IFrame for each parallel call—you don’t want to change an IFrame’s source while it’s waiting for a response. Second, you need to investigate and test limitations, because the application can grind to a halt if too many calls are outstanding.

There’s one last problem with IFrame Call: browser UIs aren’t really built with this application in mind, so some things might seem funny to users. In particular, Alex Russell has pointed out two issues that arise with IFrame Calls under IE: the “phantom click” and the “throbber of doom” (<http://alex.dojotoolkit.org/?p=538>). The former repeats a clicking sound every time an IFrame request takes place, and the latter keeps animating the throbber icon while the IFrame is loading. These devices are great for a full page refresh, like when you click on a link, but they suck when a script is trying to quietly talk with the server behind the scenes. In Alex’s article, he explains how he unearthed a fantastic hack inside Gmail’s (<http://gmail.com>) chat application. The hack suppresses the annoying behavior by embedding the IFrame inside an obscure ActiveX component called “htmlfile,” which is basically

a separate web page. As the IFrame isn't directly on the web page anymore, IE no longer clicks and throbs. See Alex's blog post (<http://alex.dojotoolkit.org/?p=538>) for full details.

Make no mistake: IFrame Call is a pure hack. IFrames were never intended to facilitate browser-server communication. It's likely the technique will slowly fade away as XMLHttpRequest gains popularity and as legacy browsers are retired. Nevertheless, it does retain a few advantages over *XMLHttpRequest Call*, mentioned in the "Alternatives" section of that pattern.

Real-World Examples

Google Maps

Google Maps (<http://maps.google.com>), perhaps the most famous Ajax App, lets you navigate through a map of the world (Figure 6-7). Location and metadata is required from the server, and this content is retrieved using IFrames. Google also uses XMLHttpRequest for downloading XSLT stylesheets (a technique described in *Browser-Side XSLT* [Chapter 11]).

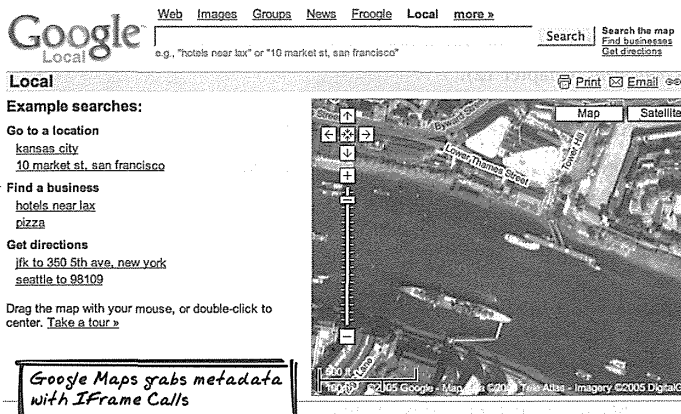


FIGURE 6-7. Google Maps

Scoop framework, Kuro5hin

Scoop (<http://scoop.kuro5hin.org>), an open source content management system, uses IFrames for discussion areas (Figure 6-8). Using the "dynamic comments" module, users can drill down a discussion thread without reloading the whole page. Scoop powers the popular Kuro5hin web site (<http://kuro5hin.org>) among others.

PXL8 Demo

Michele Tranquilli has an excellent tutorial on IFrames with embedded examples (<http://www.pxl8.com/iframes.html>).

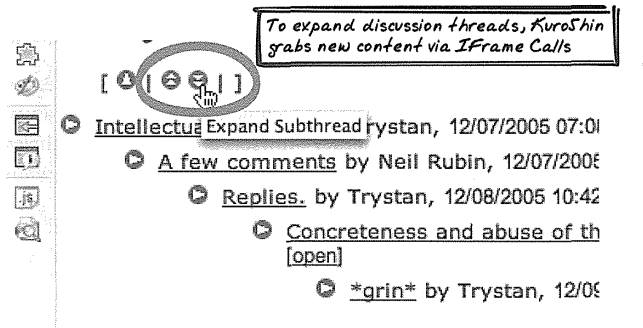


FIGURE 6-8. KuroShin

HTMLHttpRequest Library

Angus Turnbull's HTMLHttpRequest (<http://www.twinhelix.com/javascript/htmlhttprequest/>) is a portable web-remoting library that initially attempts to use XMLHttpRequest, and if that fails, falls back to IFrame. The tool is documented in the Code Examples section of *Cross-Browser Component* (Chapter 12).

Code Refactoring: AjaxPatterns Sum Demo

The Basic Sum Demo (<http://ajaxify.com/run/sum>) uses XMLHttpRequest, and there's a version available that's been refactored to use IFrame (<http://ajaxify.com/run/sum/iframe>). There's no walkthrough here, as it's very similar to the GET component of the demo discussed earlier in the "Solution."

Alternatives

XMLHttpRequest Call

XMLHttpRequest Call (see earlier in this chapter) is the natural alternative to IFrame Call. The "Alternatives" section of that pattern, earlier in this chapter, compares the two approaches.

Frame Call

Before IFrames, there were plain old frames. As a user-interface concept, they've had a few public relations problems over the years, but they do at least provide a means of remote scripting similar to IFrame Calls (even if it was accidental). Since IFrames are now widely supported, it's unlikely you'll ever need to use frames for remote scripting (or anything else, for that matter).

Metaphor

A hidden IFrame is an invisible friend you can delegate queries to.

HTTP Streaming



Connection, Duplex, Live, Persistent, Publish, Push, RealTime, Refresh, Remoting, RemoteScripting, Stateful, ReverseAjax, Stream, TwoWay, Update

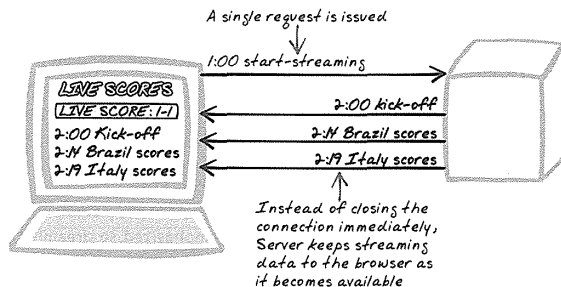


FIGURE 6-9. HTTP Streaming

Goal Story

Tracy's trading application contains a section showing the latest announcements. It's always up-to-date because the announcements are streamed directly from the server. Suddenly, an important announcement appears, which triggers her to issue a new trade. The trade occurs with a new XMLHttpRequest, so the announcement stream continues unaffected.

Problem

How can the server initiate communication with the browser?

Forces

- The state of many Ajax Apps is inherently volatile. Changes can come from other users, external news and data, completion of complex calculations, and triggers based on the current time and date.
- HTTP connections can only be created within the browser. When a state change occurs, there's no way for a server to create a physical connection to notify any interested client.

Solution

Stream server data in the response of a long-lived HTTP connection. Most web services do some processing, send back a response, and immediately exit. But in this pattern, they keep the connection open by running a long loop. The server script uses event registration or some other technique to detect any state changes. As soon as a state change occurs, it pushes new data to the outgoing stream and flushes it, but doesn't actually close it. Meanwhile, the browser must ensure the user interface reflects the new data. This pattern discusses a couple of techniques for Streaming HTTP, which I refer to as "page streaming" and "service streaming."

IS IT PRACTICAL?

Note that this pattern, though it's used in production systems, remains somewhat experimental, especially the *service-streaming* variant. There are issues of feasibility, scalability, and browser portability. One particular gotcha is the effect of proxies. Sometimes, a proxy sitting somewhere between server and browser will buffer responses, an unfortunate optimization that prevents real-time data from flowing into the browser. Even worse, some proxies will, with the best of intentions, take it upon themselves to close the connection after an idle period. Another issue is server performance, given the number of persistent connections that must be handled.

HTTP Streaming works best when you can control factors such as the web server, the network, and even the browser where possible. Scalability is often over-emphasized, so streaming might be practical on more systems than one might expect. Furthermore, some modern servers (e.g., Twisted, at <http://twistedmatrix.com/>) are being developed with streaming-style architecture in mind and are able to maintain many long-lived connections.

Page streaming involves streaming the original page response (Figure 6-10). Here, the server immediately outputs an initial page and flushes the stream, but keeps it open. It then proceeds to alter it over time by outputting embedded scripts that manipulate the DOM. The browser's still officially writing the initial page out, so when it encounters a complete `<script>` tag, it will execute the script immediately. A simple demo is available at <http://ajaxify.com/run/streaming/>.

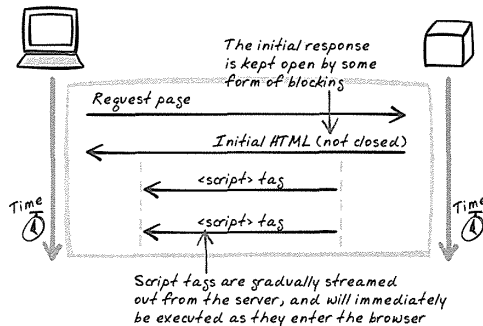


FIGURE 6-10. Page streaming

For example, the server can initially output a `div` that will always contain the latest news:

```
print("<div id='news'></div>");
```

But instead of exiting, it starts a loop to update the item every 10 seconds (ideally, there would be an interrupt mechanism instead of having to manually pause):

```
<?
    while (true) {
?>
    <script type="text/javascript">
        $('news').innerHTML = '<?= getLatestNews() ?>';
    </script>
?>
    flush(); // Ensure the Javascript tag is written out immediately
    sleep(10);
    }
?>
```

DON'T FORGET TO FLUSH: A PHP TIP

Each language and environment will have its own idiosyncrasies in implementing this pattern. For PHP, there's fortunately some very useful advice in the `flush()`: online comments (<http://php.net/flush>). It turned out to be necessary to execute `ob_end_flush()`; before `flush()` could be called. There's also a `max_execution_time` parameter you might need to increase, and the web server will have its own timeout-related parameters to tweak.

That illustrates the basic technique, and there are some refinements discussed next and in the "Decisions" section, later in this chapter. One burning question you might have is how the browser initiates communication, since the connection is in a perpetual response state. The answer is to use a "back channel"; i.e., a parallel HTTP connection. This can easily be accomplished with an *XMLHttpRequest Call* or an *IFrame Call*. The streaming service will be able to effect a subsequent change to the user interface, as long as it has some means of detecting the call—for example, a session object, a global application object (such as the `applicationContext` in a Java Servlet container), or the database.

Page streaming means the browser discovers server changes almost immediately. This opens up the possibility of real-time updates in the browser, and allows for bi-directional information flow. However, it's quite a departure from standard HTTP usage, which leads to several problems. First, there are unfortunate memory implications, because the JavaScript keeps accumulating and the browser must retain all of that in its page model. In a rich application with lots of updates, that model is going to grow quickly, and at some point a page refresh will be necessary in order to avoid hard drive swapping or worse. Second, long-lived connections will inevitably fail, so you have to prepare a recovery plan. Third, most servers can't deal with lots of simultaneous connections. Running multiple scripts is certainly going to hurt when each script runs in its own process, and even in more sophisticated multithreading environments, there will be limited resources.

Another problem is that JavaScript must be used, because it's the only way to alter page elements that have already been output. In its absence, the server could only communicate by appending to the page. Thus, browser and server are coupled closely, making it difficult to write a rich Ajaxy browser application.

Service streaming is a step towards solving these problems (Figure 6-11). The technique relies on *XMLHttpRequest Call* (or a similar remoting technology like *IFrame Call*). This time, it's an *XMLHttpRequest* connection that's long-lived, instead of the initial page load. There's more flexibility regarding length and frequency of connections. You could load the page normally, then start streaming for 30 seconds when the user clicks a button. Or you could start streaming once the page is loaded, and keep resetting the connection every 30 seconds. Flexibility is valuable, given that HTTP Streaming is constrained by the capabilities of the server, the browsers, and the network.

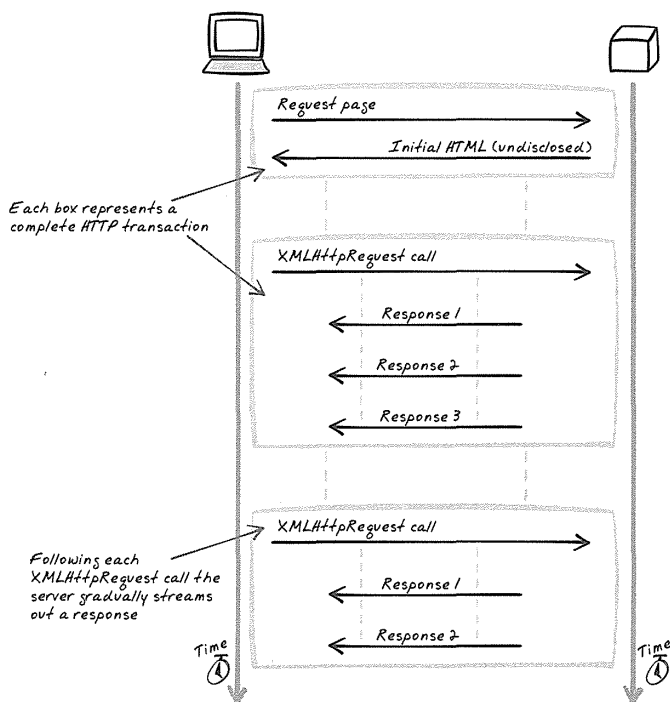


FIGURE 6-11. Service streaming

As for the mechanics of service streaming, the server uses the same trick of looping indefinitely to keep the connection open, and periodically flushing the stream. The output can no longer be HTML script tags, because the web browser wouldn't automatically execute them, so how does the browser deal with the stream? The answer is that it polls for the latest response and uses it accordingly.

The `responseText` property of `XMLHttpRequest` always contains the content that's been flushed out of the server, even when the connection's still open. So the browser can run a periodic check; e.g., to see if its length has changed. One problem, though, is that, once flushed, the service can't undo anything its output. For example, the `responseText` string arising from a timer service might look like this: "12:01:00 12:01:05 12:01:10," whereas it would ideally be just "12:01:00," then just "12:01:05," then just "12:01:10." The solution is to parse the response string and look only at the last value—to be more precise, the last *complete* value, since it's possible the text ends with a partial result. An example of this technique (<http://www.ajaxify.com/run/streaming/xmlHttpRequest/countdown/>) works in this way. To ease parsing, the service outputs each message delimited by a special token, `@END@` (an XML tag would be an alternative approach). Then, a regular expression can be run to grab the latest message, which must be followed by that token to ensure it's complete:

```
function periodicXHRCheck() {
    var fullResponse = util.trim(xhrReq.responseText);
    var responsePatt = /^(*@END@)*(.)@END@.*$/;
    if (fullResponse.match(responsePatt)) { // At least one full response so far
        var mostRecentDigit = fullResponse.replace(responsePatt, "$2");
        $("response").innerHTML = mostRecentDigit;
    }
}
```

That's great if you only care about the last message, but what if the browser needs to log all messages that came in? Or process them in some way? With a polling frequency of 10 seconds, the previous sequence would lead to values being skipped; the browser would skip from 12:01 to 12:01:10, ignoring the second value. If you want to catch *all* messages, you need to keep track of the position you've read up to. Doing so lets you determine what's new since the previous poll, a technique used in "Code Refactoring: AjaxPatterns Streaming Wiki," later in this chapter.

In summary, service streaming makes streaming more flexible, because you can stream arbitrary content rather than JavaScript commands, and because you can control the connection's lifecycle. However, it combines two concepts that aren't consistent across browsers—`XMLHttpRequest` and HTTP Streaming—with predictable portability issues. Experiments suggest that the *page-streaming* technique does work on both IE and Firefox (<http://ajaxify.com/run/streaming/>), but service streaming only works properly on Firefox, whether `XMLHttpRequest` (<http://ajaxify.com/run/streaming/xmlHttpRequest/>) or `IFrame` (<http://ajaxify.com/run/streaming/xmlHttpRequest/iframe/>) is used. In both cases, IE suppresses the response until it's complete. You could claim that's either a bug or a feature; but either way, it works against HTTP Streaming. So for portable page updates, you have a few options:

- Use a hybrid (<http://ajaxlocal/run/streaming/xmlHttpRequest/iframe/scriptTags/>) of page streaming and `IFrame`-based service streaming, in which the `IFrame` response outputs script tags, which include code to communicate with the parent document (e.g., `window.parent.onNewData(data);`). As with standard page streaming, the scripts will be executed immediately. It's not elegant, because it couples the remote service to the browser script's structure, but it's a fairly portable and reliable way to achieve HTTP Streaming.

- Use a limited form of service streaming, where the server blocks until the first state change occurs. At that point, it outputs a message and exits. This is not ideal, but certainly feasible (see “Real-World Examples” later in this chapter).
- Use page streaming.
- Use *Periodic Refresh* (Chapter 10) instead of HTTP Streaming.

Decisions

How long will you keep the connection open?

It’s impractical to keep a connection open forever. You need to decide on a reasonable period of time to keep the connection open, which will depend on:

- The resources involved: server, network, browsers, and supporting software along the way.
- How many clients will be connecting at any time. Not just averages, but peak periods.
- How the system will be used—how much data will be output, and how the activity will change over time.
- The consequences of too many connections at once. For example, will some users miss out on critical information? It’s difficult to give exact figures, but it seems fair to assume a small intranet application could tolerate a connection of minutes or maybe hours, whereas a public dotcom might only be able to offer this service for quick, specialized situations, if at all.

How will you decide when to close the connection?

The web service has several ways to trigger the closing of a connection:

- A time limit is reached.
- The first message is output.
- A particular event occurs. For example, the stream might indicate the progress of a complex calculation (see *Progress Indicator* [Chapter 14]) and conclude with the result itself.
- Never. The client must terminate the connection.

How will the browser distinguish between messages?

As the “Solution” mentions, the service can’t erase what it’s already output, so it often needs to output a succession of distinct messages. You’ll need some protocol to delineate the messages; e.g., the messages fit a standard pattern, the messages are separated by a special token string, or the messages are accompanied by some sort of metadata—for example, a header indicating message size.

Real-World Examples

LivePage

LivePage (<http://twisted.sourceforge.net/TwistedDocs-1.2.0/howto/livepage.html>) is part of Donovan Preston's Nevow framework (<http://nevow.com/>), a Python-based framework, built on the Twisted framework (<http://twistedmatrix.com/>). Events are pushed from the server using XMLHttpRequest-based service streaming. For compatibility reasons, Nevow uses the technique mentioned in the "Solution" in which the connection closes after first output. Donovan explained the technique to me:

When the main page loads, an XHR (XMLHttpRequest) makes an "output conduit" request. If the server has collected any events between the main page rendering and the output conduit request rendering, it sends them immediately. If it has not, it waits until an event arrives and sends it over the output conduit. Any event from the server to the client causes the server to close the output conduit request. Any time the server closes the output conduit request, the client immediately reopens a new one. If the server hasn't received an event for the client in 30 seconds, it sends a noop (the javascript "null") and closes the request.

Jotspot Live

Jotspot Live (<http://jotlive.com/>) is a live, multiuser wiki environment that uses HTTP Streaming to update message content (Figure 6-12). In an interview with Ajaxian.com (http://www.ajaxian.com/archives/2005/09/jotspot_live_li.html), developer Abe Fettig explained the design is based on LivePage (see the previous example).

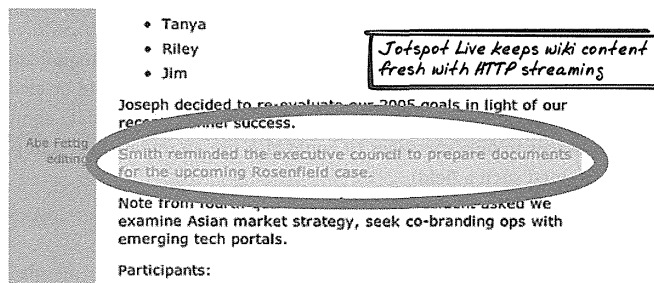


FIGURE 6-12 .JotSpot Live

Realtime on Rails

Martin Scheffler's Realtime on Rails (<http://www.uni-weimar.de/~scheffl2/wordpress/?p=19>) is a real-time chat application that uses service streaming on Firefox and, because of the restrictions described in the earlier "Solution," *Periodic Refresh* on other browsers.

Lightstreamer engine

Lightstreamer (<http://www.lightstreamer.com/>) is a commercial “push engine” used by companies in finance and other sectors to perform large-scale HTTP Streaming. Because it works as a standalone server, there are many optimizations possible; the company states that 10,000 concurrent users can be supported on a standard 2.4GHz Pentium 4 (<http://www.softwareas.com/http-streaming-an-alternative-to-polling-the-server#comment-3078>).

Pushlets framework

Just van den Broecke’s Pushlets framework (<http://www.pushlets.com/doc/whitepaper-s4.html>) is a Java servlet library based on HTTP Streaming that supports both page-streaming and service-streaming mechanisms.

Code Refactoring: AjaxPatterns Streaming Wiki

The Basic Wiki Demo (<http://ajaxify.com/run/wiki>) updates messages with *Periodic Refresh*, polling the server every five seconds. The present Demo (<http://ajaxify.com/run/wiki/streaming>) replaces that mechanism with service streaming. The AjaxCaller library continues to be used for uploading new messages, which effectively makes it a back-channel.

The *Web Service* remains generic, oblivious to the type of client that connects to it. All it has to do is output a new message each time it detects a change. Thus, it outputs a stream like this:

```
<message>content at time zero</message>
<message>more content a few seconds later</message>
<message>even more content some time after that </message>
... etc....
```

To illustrate how the server can be completely independent of the client application, it’s up to the client to terminate the service (in a production system, it would be cleaner for the service to exit normally; e.g., every 60 seconds). The server detects data changes by comparing old messages to new messages (it could also use an update timestamp if the message table contained such a field):

```
while (true) {
    ...
    foreach ($allIds as $messageId) {
        ...
        if ($isNew || $isChanged) {
            print getMessageXML($newMessage); // prints "<message>...</message>"
            flush();
        }
    }
    sleep(1);
}
```

The browser sets up the request using the standard `open()` and `send()` methods. Interestingly, there’s no `onreadystatechange` handler because we’re going to use a timer to poll the

text. (In Firefox, a handler might actually make sense, because `onreadystatechange` seems to be called whenever the response changes.)

```
xhReq.open("GET", "content.phtml", true);
xhReq.send(null);
// Don't bother with onreadystatechange - it shouldn't close
// and we're polling responsetext anyway
...
pollTimer = setInterval(pollLatestResponse, 2000);
```

`pollLatestResponse()` keeps reading the outputted text. It keeps track of the last complete message it detected using `nextReadPos`. For example, if it's processed two 500-character messages so far, `nextReadPos` will be 1001. That's where the search for a new message will begin. Each complete message after that point in the response will be processed sequentially. Think of the response as a work queue. Note that the algorithm doesn't assume the response ends with `</message>`; if a message is half-complete, it will simply be ignored.

```
function pollLatestResponse() {
    var allMessages = xhReq.responseText;
    ...
    do {
        var unprocessed = allMessages.substring(nextReadPos);
        var messageXMLEndIndex = unprocessed.indexOf("</message>");
        if (messageXMLEndIndex != -1) {
            var endOffFirstMessageIndex = messageXMLEndIndex + "</message>".length;
            var anUpdate = unprocessed.substring(0, endOffFirstMessageIndex);
            renderMessage(anUpdate);
            nextReadPos += endOffFirstMessageIndex;
        }
    } while (messageXMLEndIndex != -1);
}
```

After some time, the browser will call `xhReq.abort()`. In this case, it's the browser that stops the connection, but as mentioned earlier, it could just as easily be the server.

Finally, note that the uploading still uses the `AjaxCaller` library. So if the user uploads a new message, it will soon be streamed out from the server, which the browser will pick up and render.

Alternatives

Periodic Refresh

Periodic Refresh (Chapter 10) is the obvious alternative to HTTP Streaming. It fakes a long-lived connection by frequently polling the server. Generally, Periodic Refresh is more scalable and easier to implement in a portable, robust manner. However, consider HTTP Streaming for systems, such as intranets, where there are fewer simultaneous users, you have some control over the infrastructure, and each connection carries a relatively high value.

TCP connection

Using a *Richer Plugin* like Flash or Java, the browser can initiate a TCP connection. The major benefit is protocol flexibility—you can use a protocol that's suitable for long-lived connections, unlike HTTP, whose stateless nature means that servers don't handle long connections very well. One library that facilitates this pattern is Stream (<http://www.stormtide.ca/Stream/>), consisting of an invisible Flash component and a server, along with a JavaScript API to make use of them.

Related Patterns

Distributed Events

You can use *Distributed Events* (Chapter 10) to coordinate browser activity following a response.

Metaphor

Think of a “live blogger” at a conference, continuously restating what's happening each moment.

Want To Know More?

- Donovan Preston on his creation of LivePage (http://ulaluma.com/pyx/archives/2005/05/multiuser_prog.html)
- Alex Russell on Comet (<http://alex.dojotoolkit.org/?p=545>)

Acknowledgments

Thanks to Donovan Preston and Kevin Arthur for helping to clarify the pattern and its relationship to their respective projects, LivePage and Stream.

On-Demand JavaScript



Behaviour, Bootstrap, CrossDomain, Dynamic, JavaScript, LazyLoading, OnDemand, ScriptTag

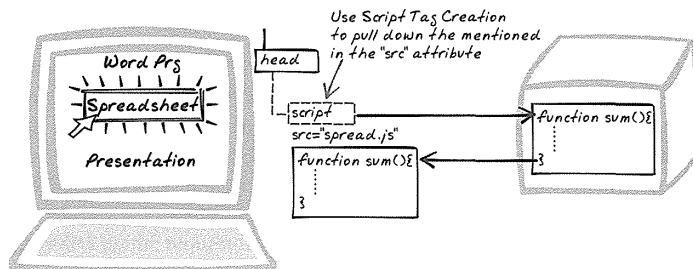


FIGURE 6-13. On-Demand JavaScript

Goal Story

Bill's logging into his bank web site. The login form comes up straight away, and as he types in his username and password, the browser is quietly downloading the JavaScript necessary for the rest of the application.

Problem

How can you deploy lots of JavaScript code?

Forces

- Ajax Apps make heavy use of JavaScript. Richer browser behavior means bulkier JavaScript to download.
- Downloading JavaScript has a performance impact. Interaction cannot fully begin until all initial JavaScript has been loaded.
- Bandwidth is also a concern with heavy JavaScript content. Often, not all JavaScript that's loaded is actually used, leading to a waste of bandwidth.

Solution

Download and run JavaScript snippets. The initial page load includes some JavaScript code, which—among other things—contains the bootstrapping code necessary to pull down further JavaScript. There are two techniques available: *Script Tag Creation* and *Service Eval*. Each will now be described, following by three main applications of the pattern.

With *Script Tag Creation*, use DOM manipulation to inject a new script element into the page. Perhaps surprisingly, the effect will be exactly the same as if the `<script>` tag had been encountered on startup: the referenced JavaScript will be downloaded and executed automatically. The tag can be attached to either the head or the body, though the former is more common as that's where you'd usually find script tags:

```
var head = document.getElementsByTagName("head")[0];
script = document.createElement('script');
script.id = 'importedScriptId';
script.type = 'text/javascript';
script.src = "http://path.to.javascript/file.js";
head.appendChild(script);
```

How will you know when the script has loaded? It seems browsers vary in their behavior here, with some loading the script synchronously and some not. In IE, you can be notified with a mechanism similar to XMLHttpRequest's (<http://www.xml.com/lpt/a/2005/11/09/fixing-ajax-xmlhttprequest-considered-harmful.html#onreadystatechange>). With other browsers, you might need to keep polling for some indication it's loaded (where the indication depends on what the script is meant to do). If you happen to control the server script, you could implement a notification mechanism; i.e., complete the script by calling back to a listener,

»

if it exists. There's actually a promising proposal, JSONP (<http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/>), which aims to have a simple, flexible mechanism like this, standard across the entire industry.

Script Tag Creation doesn't let you retrieve any old text; it has to be valid JavaScript. And your script won't be able to read it directly because the browser will use it only for evaluation. So how can you get hold of some data from an external server? The most common way is for the remote JavaScript to assign a variable to the required data:

```
var characters = new Array("Mario", "Sonic", "Lara");
```

The content is usually a data structure, just like a standard *JSON Message*, but it requires an assignment as well, in order to be referenced in the code. It could also be a function that returns the desired value. Either way, the browser code has to use whatever name is mentioned in the script, which isn't ideal. Again, the idea of a flexible script mechanism like J is worth considering, as it would let the caller decide on the variable name.

Service Eval is the other technique for On-Demand JavaScript, though not as prominent as *Script Tag Creation* for reasons we'll discuss later. Under *Service Eval*, a *Web Service* is called with a standard *XMLHttpRequest Call*, it outputs some JavaScript as response content, and the JavaScript is then executed with an `eval()` call. We're just inspecting the *XMLHttpRequest Call's* `responseText` property, which we could manipulate before evaluating, so the body doesn't have to be a complete, valid piece of JavaScript (unlike with *Script Tag Creation*).

Any code not inside a function will be executed immediately. To add new functionality for later on, the JavaScript can add directly to the current window or to an object that's already known to exist. For example, the following can be sent:

```
self.changePassword = function(oldPassword, newPassword) {  
    ...  
}
```

The *XMLHttpRequest* callback function just needs to treat the response as plain-text and pass it to `eval()`. A warning: here again, asynchronicity rears its ugly head. You can't assume the new code will be available immediately after requesting it, so don't do this:

```
if (!self.changePassword) {  
    requestPasswordModuleFromServer();  
}  
changePassword(old, new) // Won't work the first time because  
                          // changePassword's not loaded yet.
```

Instead, you either need to make a synchronous call to the server, add a loop to keep checking for the new function, or explicitly make the call in the response handler.

On-Demand JavaScript has three distinct applications:

Lazy Loading

Defer loading of bulky JavaScript code until later on. Works with either On-Demand JavaScript technique (*Service Eval* or *Script Tag Creation*).

Behavior Message

Have the server respond with a kind of “Behavior Message,” which dictates the browser’s next action. Works with either On-Demand JavaScript technique (*Service Eval* or *Script Tag Creation*).

Cross-Domain Scripting

Using *Script Tag Creation*, bypass the standard “same-origin” policy that normally necessitates a *Cross-Domain Proxy*. Works only with *Script Tag Creation*.

Let’s look at Lazy Loading first. Conventionally, best practice has been to avoid including JavaScript unobtrusively—by including it in one or more script tags:

```
<html>
  <head>
    <script type="text/javascript" src="search.js"></script>
    <script type="text/javascript" src="validation.js"></script>
    <script type="text/javascript" src="visuals"></script>
  </head>
  ...
</html>
```

Lazy Loading builds on this approach to suggest just a minimal initialization module in the initial HTML:

```
<html>
  <head>
    <script type="text/javascript" src="init.js"></script>
  </head>
  ...
</html>
```

The initialization module declares whatever actions are required to start up the page and perhaps enough to cover typical usage. In addition, it must perform a bootstrapping function, pulling down new JavaScript on demand.

The second application, Behavior Message, is a variant of the *HTML Message* pattern. Whereas Lazy Loading sets up library code for ongoing use, Behavior Messages take some transient code and runs `eval()` on it immediately—the script is not wrapped inside a function (although it may define some functions that it calls). Effectively, the browser is asking the server what to do next.

Cross-Domain Scripting is the third application of this pattern. script tags have always been able to include source JavaScript from external domains. The rule not only applies to static `<script>` tags, but also to dynamically created script tags as in the *Script Tag Creation* technique. Thus, unlike with `XMLHttpRequest` and `IFrame`, your script can directly access external content this way. And because the `src` property can be any URL, you can pass in arguments as CGI variables. The idea is becoming rather popular, with companies such as Yahoo! offering JavaScript APIs specifically for this approach (see “Real-World Examples,” later in this section).

Running a script from an external domain can be useful when you trust it, and ideally control it. Other times, it's a definite security risk. Douglas Crockford, creator of JSON warns of the havoc an external script can wreak (<http://www.mindsack.com/uxe/dynodes/>) (emphasis mine):

That script can deliver the data, but it runs with the same authority as scripts on the base page, so it is able steal cookies or misuse the authorization of the user with the server. A rogue script can do destructive things to the relationship between the user and the base server.... **The unrestricted script tag hack is the last big security hole in browsers.** It cannot be easily fixed because the whole advertising infrastructure depends on the hole. Be very cautious.

Decisions

Will you use Service Eval or Script Tag Creation?

The choice is between *Service Eval* and *Script Tag Creation* depends on several factors. *Service Eval* has the following benefits over *Script Tag Creation*:

- Being based on XMLHttpRequest, there's a standard mechanism for being notified when the script is ready, so there's no risk of calling functions that don't yet exist.
- You get access to the raw script code.
- There's more flexibility on the message format: you can, for example, send several JavaScript snippets inside different XML nodes, and have the browser script extract them out.

And *Script Tag Creation* has a couple of benefits over *Service Eval*:

- You can load JavaScript from external domains. This is the only significant functional difference between the two styles.
- The JavaScript will automatically be evaluated in much the same way as the JavaScript linked in the static HTML is evaluated when the `<script>` tag is first encountered. Thus, you don't have to explicitly add variables and functions to the document in order to use them later; you just declare them normally.

With Lazy Loading, how will you break modules down?

You'll need to decide how to carve up your JavaScript. Standard principles of software development apply: a module should be well-focused, and intermodule dependencies should be avoided where possible. In addition, there are web-specific concerns:

- Ideally, any given module is either not used at all, or used in entirety. What you don't want is a 5000-line module that's downloaded to retrieve a 3-line function. It's a waste of bandwidth that defeats the main purpose of On-Demand JavaScript.
- You need one or more modules present on startup. At least one is required to kick off further downloads as required.

- Keep in mind that code will probably be cached locally. Alexander Kirk has done some experimentation with caching of On-Demand JavaScript (<http://alexander.kirk.at/2005/10/11/caching-of-downloaded-code-testing-results/>), and it turns out that all major browsers will cache code created with *Script Tag Generation*. You can usually ensure that responses from XMLHttpRequest are also cached.

With Lazy Loading, at what stage will the script download the JavaScript?

It's easiest to download the JavaScript just before it's required, but that's not always the best approach. There will be some delay in downloading the JavaScript, which means the user will be waiting around if you grab it at the last possible moment. When the user's actions or the system state suggest some JavaScript will soon be needed, consider downloading it immediately.

Predicting if JavaScript is needed is an example of *Predictive Fetch* (Chapter 13)—grabbing something on the hunch that it might be needed. You need to make a trade-off about the likelihood it's required against the hassle caused if you hold off until later. For example, imagine you have some JavaScript to validate a completed form. If you wait until the end, you'll definitely not be wasting bandwidth, but it's at the expense of the user's satisfaction. You could download it when there's one field to go, or two fields, or when the form's first loaded. Each option increases the chance of a wasted download, but increases the chance of a smoother validation procedure.

Real-World Examples

MapBuilder

MapBuilder (<http://mapbuilder.sourceforge.net>) is a framework for mapping web sites (Figure 6-14). It uses On-Demand JavaScript to reduce the amount of code being downloaded. The application is based on a Model-View-Controller paradigm. Model information is declared in an XML file, along with the JavaScript required to load corresponding widgets. When the page starts up, only the required JavaScript is downloaded, instead of the entire code base. This is therefore a partial implementation of this pattern; it does ensure that only the minimal subset of JavaScript is ever used, but it doesn't load pieces of the code in a lazy, on-demand style.

Delicious/Yahoo! APIs

Social bookmarking web site Delicious (<http://del.icio.us>), and its owner, Yahoo!, both offer JSON-based APIs, with appropriate hooks to allow direct access from the browser via *Script Tag Creation*. The Delicious API (<http://del.icio.us/help/json>) will create a new object with the result (or populate it if it's already present). The Yahoo! API (<http://developer.yahoo.net/common/json.html>) allows you to specify a callback function in your script that the JSON will be passed to.

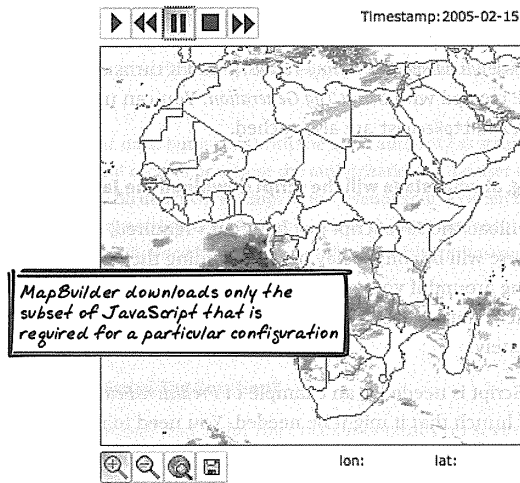


FIGURE 6-14. MapBuilder

Dojo packaging framework

Dojo (<http://dojotoolkit.org/download>) is a comprehensive framework aiming to simplify JavaScript development. As such, it provides a number of scripts, of which an individual project might only use a subset. To manage the scripts, there's a Java-like package system, which lets you pull in new JavaScript as required (<http://dojo.jot.com/WikiHome/Documents/DojoPackageSystem>). You need only include a single JavaScript file directly:

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

You then pull in packages on demand with the Dojo API:

```
dojo.hostenv.moduleLoaded("dojo.aDojoPackage.*");
```

Running the above command will cause Dojo to automatically download the modules under `dojo.aDojoPackage.*`.

JSAN import system

The JavaScript Archive Network (JSAN) (<http://openjsan.org>) is an online repository of scripts. As well, it includes a library for importing JavaScript modules, also a convention similar to Java. The following call:

```
JSAN.use('Module.To.Include');
```

* It's possible not all modules will be downloaded, because the precise set can be defined by the package author and can be made dependent on the calling environment (whether browser or command line).

is mapped to *Module/To/Include.js*. JSAN.*includePath* defines all the possible top-level directories where this path can reside. If the *includePath* is `["/", "/js"]`, JSAN will look for */Module/To/Include* and */js/Module/To/Include*.

Code Example, AjaxPatterns On-Demand JavaScript Wiki

Introducing On-Demand JavaScript to the Wiki Demo

In the Basic Wiki Demo (<http://ajaxify.com/run/wiki>), all JavaScript is downloaded at once. But many times, users will only *read* from the wiki—why download the code to *write* to it? So this demo refactors to On-Demand JavaScript in three stages:

1. The *uploadMessage* function is extracted to a second JavaScript file, *upload.js*. There's no On-Demand JavaScript yet because both files are included.
2. A further refactoring introduces On-Demand JavaScript by ensuring that *upload.js* file is only downloaded if and when an upload occurs. This version uses *Script Tag Creation*.
3. In yet another refactoring, the *Script Tag Creation* technique is replaced with *Service Eval*.

Separate JavaScript: Extracting upload.js

In the first refactoring (<http://ajaxlocal/run/wiki/separate.JS/>), the *upload* function is simply moved to a separate JavaScript file. The initial HTML includes the new file:

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

The new *upload.js* now contains the *uploadMessage* function. Reflecting the separation, a couple of parameters are introduced to help decouple the function from the main wiki script:

```
function uploadMessages(pendingMessages, messageResponseHandler) {
    ...
}
```

The calling code is almost the same as before:

```
uploadMessages(pendingMessages, onMessagesLoaded);
```

We've thus far gained a bit of modularity, but don't have a boost to performance yet, since both files must be downloaded on startup.

Script Tag Creation On-Demand JavaScript

With On-Demand JavaScript, the *upload.js* is no longer required on startup, so its reference no longer appears in the initial HTML, leaving just the main module, *wiki.js*:

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

A new function has been added to download the script. To avoid downloading it multiple times, a guard clause checks if `uploadMessages` already exists, and if so, immediately returns. Following the *Script Tag Creation* technique, it adds a script element to the document's head, initialized with the `upload.js` URL and a standard JavaScript type attribute:

```
function ensureUploadScriptIsLoaded() {
  if (self.uploadMessages) { // Already exists
    return;
  }
  var head = document.getElementsByTagName("head")[0];
  script = document.createElement('script');
  script.id = 'uploadScript';
  script.type = 'text/javascript';
  script.src = "upload.js";
  head.appendChild(script);
}
```

The calling script just has to invoke this function. However, as mentioned earlier in the “Solution,” it's possibly downloaded asynchronously, so a check must be made here too:

```
ensureUploadScriptIsLoaded();
if (self.uploadMessages) { // If not loaded yet, wait for next sync
  uploadMessages(pendingMessages, onMessagesLoaded);
  ....
}
```

If scripts are loaded asynchronously by the browser, the test will actually fail the first time, because the download function will return before the script's been downloaded. But in this particular application, that doesn't actually matter much—the whole synchronization sequence is run every five seconds anyway. If the upload function isn't there yet, it should be there in five seconds, and that's fine for our purposes.

Service Eval On-Demand JavaScript

The *Script Tag Creation* code is refactored here to use an *Service Eval* instead, where an *XMLHttpRequest Call* retrieves `upload.js` and evals it. The initial script—`wiki.js` differs only in its implementation of the JavaScript retrieval. The text response is simply passed to eval:

```
function ensureUploadScriptIsLoaded() {
  if (self.uploadMessages) { // Already exists
    return;
  }
  ajaxCaller.getPlainText("upload.js", function(jsText) { eval(jsText); });
}
```

The JavaScript response must also change. If it simply defined a global function like before; i.e.:

```
function uploadMessages(pendingMessages, messageResponseHandler) {
  ...
}
```

then the function would die when the evaluation completes. Instead, we can achieve the same effect by declaring the function as follows (this will attach the function to the current window, so it will live on after the script is evaluated).

```
uploadMessages = function(pendingMessages, messageResponseHandler) {  
    ...  
}
```

Related Patterns

HTML Message

The Behavior Message usage of this pattern is a companion to *HTML Message* (Chapter 9) and follows a similar server-centric philosophy in which the server-side dynamically controls browser activity.

Predictive Fetch

Apply *Predictive Fetch* (Chapter 13) to On-Demand JavaScript by downloading JavaScript when you anticipate it will soon be required.

Multi-Stage Download

The Lazy Loading application of this pattern is a like *Multi-Stage Download*, which also defers downloading. The emphasis in Multi-Stage Download is downloading semantic and display content rather than downloading JavaScript. In addition, that pattern is more about downloading according to a prescheduled sequence rather than downloading on demand.

Want to Know More?

- Dynamic Data Using the DOM and Remote Scripting, A Tutorial by Thomas Brattli (<http://www.dhtmlcentral.com/tutorials/tutorials.asp?id=11>)

Dynamic Behavior

NITH THE TECHNOLOGIES COVERED SO FAR, WE CAN CHANGE THE DISPLAY AND TALK TO THE SERVER. But what will trigger these actions? Ajax Apps are generally driven from within the browser, so that's where the actions will be triggered. Broadly speaking, there are two types of triggers, each covered here: *User Actions*, such as mouse clicks and keypresses, and *Scheduling*, where actions are scheduled to be run at some point in the future.

User Action



Action, Change, Click, Control, DHTML, DOM, Events, Keyboard, Mouse, Move, Type, Widget

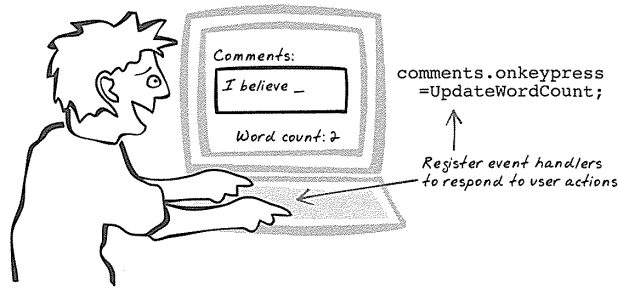


FIGURE 7-1. User Action

Goal Story

Pam is booking a trip on the corporate travel planner. She sees a form with the usual fields and clicks on location. Suddenly, a list of cities fades in beside the form, and Pam selects Paris. Beside the city list, a second list appears, this one showing approved hotels. Pam chooses the Hilton, and both lists disappear. Pam's pleased to proceed with the destination on the updated form, which now reads, "Paris Hilton."

Problem

How can the application respond to user activity?

Forces

- A rich application allows users to interact with it, frequently and in different ways.
- Responses must be as quick as possible, so as to streamline performance, keep user attention, and help the user understand the application's cause-effect model.
- Using form submissions as the only response to user activity is slow and limits interactivity.

Solution

Handle most User Actions within JavaScript, using event handlers. The essence of Ajax is rich browser-based interaction, and DOM events are the technology that make it happen. DOM objects can register event handlers, functions that are notified when events occur. This callback model should be familiar to anyone who's worked with desktop GUI frameworks.

Let's say you want to run the following function when the user clicks a shutdown button:

```
function shutdown() {
  if (confirm("Are you sure you want to shutdown?")) {
    postShutdownMessageToServer();
  }
}
```

EVENT HANDLING, BEHIND THE SCENES

If you've not worked with a modern user interface framework such as Java's Swing, you might wonder how JavaScript events work. The answer is that, behind the scenes, the operating system tells the browser what the user's doing, and the browser then passes this information to the script. For instance, the operating system—monitoring keyboard and mouse input—might notice that the user hit "X" while the keyboard focus was on the browser window. It will then notify the browser, and the browser will decide which element within the current web page has keyboard focus. If that element happens to have an associated keypress event handler, the browser will arrange for it to be invoked. Just prior to invoking it, it will construct an event object indicating the key that was pressed ("X"), the object that had focus at the time, and auxiliary information like the current mouse location and the current time. The browser will derive some of this information from the previous operating system notification, so other information might be queried there and then. When the event handler is invoked, this event object will be passed in as an argument.

User Actions are events associated closely with the DOM and use the same callback handler mechanism as many other programming environments, described by Gamma et al.'s (1995) Observer pattern. You'll see callback handlers used in other situations too:

`onload` and `onunload` are technically the same type of event as the User Actions described in this pattern because they work on DOM elements and use the same syntax. They relate to element lifecycles as opposed to user activity, which is why this pattern doesn't include them in the definition of User Actions. In most scripts, only the window (or document) `onload` is used; as this pattern shows, it's a good place to register User Actions, among other things. `onunload` is sometimes used to save data or confirm the user wants to quit.

The *XMLHttpRequest Call* (Chapter 6) pattern shows how you can register a callback to be notified of network activity as a remote call proceeds.

The *Scheduling* pattern shows how you can register a function to be executed at a certain time, or at fixed periods. There's no argument passed in, but the mechanism is similar to a DOM object event callback.

The simplest way to set this up is to declare a button with an `onclick` event handler:

```
<button id="quitButton" onclick="shutdown();" />Quit</button> <!-- Obtrusive -->
```

Now the web browser will arrange for `shutdown()` to be called whenever the button is clicked. However, we can improve on this, because the above declaration mixes JavaScript with HTML. It's cleaner to just declare the button and deal with an event handler in a separate JavaScript file. Since we always want this behavior, we should declare it as soon as the page loads. To run something when the page loads, we can use another event handler that is triggered by browser activity rather than a User Action: `onload`.

```
[HTML]
<button id="quitButton">Quit</button>

[Javascript]
window.onload = function() {
    quitButton.onclick = shutdown;
}
```

Note that we're declaring this inside `window.onload` instead of "out in the open"; if you do the latter, you might get an error because the script might be executed before the button is actually on the page. You'll see the `window.onload` idiom used in most JavaScript code, including all the Ajax Patterns demos.

Instead of referencing a callback function, it's sometimes convenient to define the callback as a closure (anonymous function), as in:

```
quitButton.onclick = function() {
    if (confirm("Are you sure you want to shutdown?")) {
        postShutdownMessageToServer();
        quitButton.onclick=null;
    }
}
```

Registering events with JavaScript, as opposed to in HTML tags, is an example of unobtrusive JavaScript because it separates JavaScript from HTML. And defining the event handler in JavaScript also has another benefit: you can dynamically redefine actions in response to system events. Our `shutdown()` method could also redefine the handler to avoid a double shutdown:

```
function shutdown() {
    if (confirm("Are you sure you want to shutdown?")) {
        postShutdownMessageToServer();
        quitButton.onclick=null; // Quit button no longer triggers an event.
    }
}
```

Notice the model here involves a single handler for any event type; the above commands *set* the handler, in a manner that will remove any existing handlers. In most cases, that's just fine, and it has the merit of being completely portable. In some situations, though, it's nice to *add* a handler instead as it makes the code more modular. Two separate library functions can then register for the same events, without having to be aware of each other. Likewise, a function for removing would also be nice:

```
addEvent(quitButton, "click", postShutdownMessageToServer);
...
removeEvent(quitButton, "click", postShutdownMessageToServer);
```

Browsers do offer support for this functionality, but it's unfortunately varied, and a portable solution has been notoriously difficult. So much so that a competition was recently held to find the best `addEvent()` and `removeEvent()` functions, and you can find the winner, a 15-line script online (http://www.quirksmode.org/blog/archives/2005/10/how_do_i_create.html). Dojo Toolkit (<http://dojotoolkit.org>) also supports this behavior as part of its sophisticated event library.

It's not always enough for the event handler to know *that* an event has occurred; it also needs to know *about* the event. For example, the same event handler might be used for three different buttons, in which case it will need to know which of the buttons was clicked. For this reason, the web browser creates an event object upon each user event, containing various bits of information. In Firefox, it's passed to the event handler, so you just ensure an event parameter exists:

```
function shutdown(ev) {  
    ...  
}
```

In previous examples, we omitted the event parameter, which is just fine since parameters are optional in JavaScript functions—omitting them just means you don't get an opportunity to use them.* As it happens, IE doesn't pass the value in anyway, and instead holds the event in a window attribute. Again, JavaScript's loose handling of parameters means you won't actually get an error by including the parameter in IE. However, the value will always be null, which isn't very useful. What all this leads to is the following boilerplate code, which you can use whenever you care about the event. An "equalizer" statement gets hold of the event whichever browser we're in:

```
function shutdown(ev) {  
    event = event || window.event;  
    ...  
}
```

The event object contains various information, such as which element was clicked and where the mouse was. The various event types are covered next.

Decisions

What events will the script listen for?

Many events are made available to JavaScript code, and more come out with each new browser upgrade. Following are some frequently used and portable events, along with typical applications. Check the following out for more info on events: http://www.quirksmode.org/js/events_compinfo.html, and <http://www.gatescript.com/events.html>.

All handler functions accept a single parameter representing the event, and as discussed earlier in the "Solution," you have two options: ignore the parameter altogether (as in the initial `shutdown()` examples), or—if you care about the event details—include the parameter and equalize it (as in the `shutdown(ev)` examples above).

Key pressing—`onkeypress`, `onkeydown`, `onkeyup`

- `onkeypress` and `onkeydown` occur immediately after a key is pressed, and will also repeat if the key is held down. `onkeyup` is called just once, upon the key's release.

* Strictly speaking, you can still read all parameter values using the special arguments array.

- They can be used to enhance standard text editing. For instance, you can confine a phone number text field to contain only numbers, or you can show a word count while the user types.
- They're sometimes used to automatically leave a field once it's valid—e.g., to proceed to the next field after five digits have been added to a zip code field. However, doing so is often counter-productive, as users generally perform faster when behavior is consistent, even at the expense of minor technical shortcuts.
- They can be used to create keyboard shortcuts for custom controls. You can determine if the user's mouse is over the control with the `onmouse*` functions, and if so, respond to particular keypresses.

WHICH KEYS TO USE FOR SHORTCUTS?

Web browsers have their own keyboard events, such as Up and Down for scrolling and the popular Ctrl-D for bookmarking. It's best to avoid using these—they'll confuse users if the browser allows them or simply fail to work if the browser forbids them. In addition, keyboard shortcuts might also conflict with operating system shortcuts such as those for cut-and-paste. This effectively means avoiding all standard shortcuts on all major operating systems. For all these reasons, the most common shortcuts are plain old keys—letters, numbers, and symbols, with no modifiers. However, that won't work while the user's editing text, so if that happens a lot, you'll need to be inventive and perhaps offer some flexibility.

Keyboard focus—onblur, onfocus

- In the case of editable fields, `onblur` indicates keyboard focus has been lost, suggesting an update has probably occurred, so is often used to initiate a remote call or some validation technique.
- `onfocus` suggests the user has begun working on a particular object, so it can be used to show online help or change the display inside a *Status Area*.

Mouse button clicking—onmouseup, onmousedown, onclick, ondblclick

- `onclick` and `ondblclick` indicate a button has been clicked or double-clicked. `onmousedown` and `onmouseup` indicate a button has been depressed or released. These latter events are more fine-grained than clicking, which implies the sequence of `mousedown` followed by `mouseup` has completed, both on the same element (note that click won't fire if the user releases the mouse button over a different element). The button control is specifically geared for catching click events to let the user do something, and `radiobuttons` and `checkboxes` can also be associated with click listeners to indicate changes.
- `onmousedown` and `onmouseup` can be used for panning behavior and for custom drag-and-drop functionality.

Mouse movement—onmouseover, onmouseout

- onmouseover and onmouseout indicate the mouse has just moved over, or has just left, an element. It can be useful to keep a pointerElement variable to track which element is currently selected.
- They can be used to change an element's style when the mouse rolls over it. This shows it's active and can convey that this is the element that will be affected if the user clicks the mouse button right now, or perhaps hits a certain key.
- They can also be used to provide help or further information in a *Status Area* or a *Popup*.

Selection—onselect

- onselect indicates when the user has selected some text.
- By tracking the selection, the application can provide information based on what the user's selected. For example, you could let the user search on a term by selecting it, and then morph the Search Results element.
- By tracking the selection, the application can also allow transformations to occur. For example, the textarea in many modern content management applications, such as mediawiki, allows the user to select some text and then change it, just like in a word processor. To italicize text on Wikipedia, select the text and click the <i> icon, which then wraps mediawiki markup (') around the selected text.

Value change—onchange

- onchange indicates a value has changed, so it's often used to initiate a remote call or some validation technique. This is an alternative to onblur. Unlike onblur, it is only called if the value is actually altered.

What attributes of the event will be inspected?

The event object contains several useful pieces of information about the event and what was going on at the time. Note that some of these attributes are set even for events you may not expect. For example, the ctrlKey modifier will be set even for a mouse-click event. This would allow you to detect a Ctrl-mouse press action. However, not all attributes are always set, so you need to be careful in testing for portability.

Following are some of the portable and more frequently used attributes of the event object:

Element—target (Firefox), srcElement (IE)

- target and srcElement indicate which element the event occurred on. To equalize across browsers:

```
el = ev.target || ev.srcElement
```

This is useful if you have a single function listening to lot of elements—for instance, an e-commerce itemClickListener monitoring all items for a click event. Inspecting this property will tell it which particular item was clicked.

Event Type—type

- type indicates which event type took place; e.g., click.
- This is a potential code issue, because it suggests the same function has been configured to handle multiple events. If it then needs to distinguish among the different types of events, it might be worth breaking it out into a handler function for each event type, with any common routines placed elsewhere.

Key code—which (Firefox), keyCode (IE)

- which and keyCode indicate the Unicode value of the key that was pressed.* This isn't completely consistent across browsers but is easy enough to equalize. Since you can't directly register a function against a specific key, this property is the only way to decide if a certain key was pressed.

Key modifiers—altKey, ctrlKey, shiftKey

- The altKey, ctrlKey, and shiftKey are modifiers indicating if the special keys Alt, Ctrl, and Shift were being held down while a key event occurred. You can use the modifiers to introduce keyboard shortcuts to the application. Since many single-modifier shortcuts are already used by one browser or another, portable applications often need to use double-modifiers. Thus, the key-handling function will need to perform a check like:

```
if (ev.ctrlKey && ev.shiftKey) {  
    ... // perform ctl-shift shortcut  
}
```

- There is also a meta-key modifier, which is generally not advisable as it's not supported by IE, and in any event, available only on certain keyboards.

Mouse buttons—button

- This indicates which mouse buttons were being associated with the event. In IE, 1 is left, 2 is right, and middle is 4. The value represents the sum of all buttons being pressed, allowing you to catch "chords"—multiple keys held down at once. In Firefox, 0 is left, 1 is middle, and 2 is right.
- This is a painful area due to serious incompatibility issues (http://www.quirksmode.org/js/events_compinfo.html). As well as the differences above, beware of incompatibilities when one button is being depressed while another is already depressed, and also incompatibilities in which events provide this property (sometimes only mouse clicks; sometimes others).

Mouse position—clientX, clientY

- These indicate the position of the mouse pointer when the event took place, relative to the browser window.
- This is useful for image-based applications, such as maps and simulators. It's often not practical to register event handlers here, so JavaScript code—with possible help of

* Find character codes with the Unicode chart at <http://www.macchiato.com/unicode/charts.html>.

web remoting—can determine exactly what the user clicked on by examining the coordinates.

Will event handlers be registered after the page has loaded?

Using JavaScript and the DOM, redefining event handlers is easy enough to do, but should you do it? Redefining the effect of user events must be done with caution, as there is great potential to confuse users. Sometimes, event redefinition occurs simply because the programmer can't be bothered adding a new control, or the UI is so small that designers want to reuse an existing control. So before deciding to redefine an event, ask yourself if there are alternatives. For example, could you add a second button instead of redefining the first button's action?

A few examples where event redefinition might be worthwhile:

- For state transitions. The JavaScript may have separate `start()` and `stop()` methods, and you need a toggle button to flip the state, since that is clearer and less error-prone than separate “on” and “off” buttons.
- For enabling and disabling. There is already disabled a available for standard controls (<http://www.quirksmode.org/js/disabled.html>), but for custom controls that you may have created, you might use event redefinition to cancel or re-enable the effects of interacting with the control.
- For actions that depend on dynamic information, such as which field has input focus.

However, in all of these cases, it's usually simpler to have a single method, always registered in the same way, and to allow that method's JavaScript to decide where to route the event.

Real-World Examples

Google Reader

Google Reader (<http://google.com/reader>) is a web-based RSS aggregator (Figure 7-2). You can change the current article by mouse-clicking on article titles. An interesting feature is keyboard shortcuts—when the page contains numerous articles, clicking “j” and “k” will scroll up and down to the previous or next story.

Google Maps

Google Maps (<http://maps.google.com>) uses a dragging action to pan the map within a *Virtual Workspace*, and the arrow keys can also be used.

Backpack

37Signals' Backpack (<http://www.backpackit.com/>) maintains items in a list and illustrates how you can use *Drag-And-Drop* in an Ajax App. *Drag-And-Drop* relies on monitoring the mouse button as well as position.

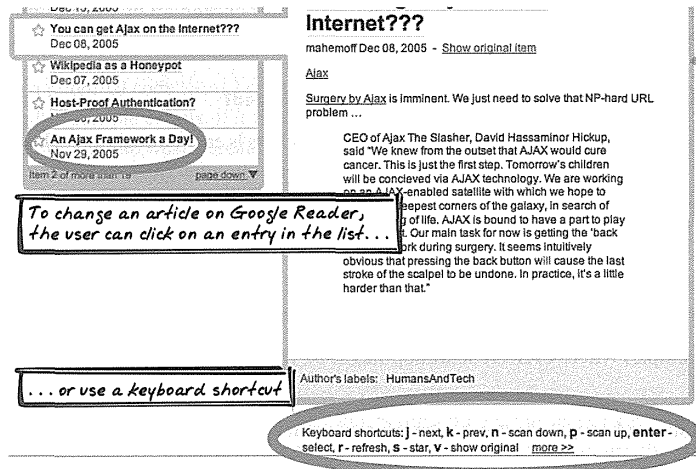


FIGURE 7-2. Google Reader

Code Example: Basic AjaxPatterns Demos

Here are a couple of basic examples from the Ajax demos. The Basic Time Demo (<http://ajaxify.com/run/time>) handles a button click like this:

```
$("#defaultTime").onclick=requestDefaultTime;
```

The wiki tracks that focus and blur events in order to show the user which message is being edited and to upload any messages after a blur occurs. It also tracks mouse movement over each area, to provide an affordance indicating that the fields can be edited:

```
messageArea.onmouseout = onMessageMouseOut;
messageArea.onmouseover = onMessageMouseOver;
messageArea.onfocus = onMessageFocus;
messageArea.onblur = onMessageBlur;
```

Each of these passes to `getMessage`, which identifies the message element that was acted upon:

```
function getMessage(event) {
    event = event || window.event;
    return event.target || event.srcElement;
}
```

Alternatives

"Click 'n' Wait"

The conventional web app follows the "click 'n' wait" pattern, popular in 1970s main-frame-based client-server applications and revived in time for the late-1990s web generation, albeit in color. The only type of interactivity is the user submitting a static form to a

server-side CGI script or clicking on a link. The script then reads some variables, does something, and outputs a whole new page of HTML. A full page refresh once in a while is OK, when a big context switch takes place, but basic updates are best controlled with JavaScript.

Richer forms

The “richer form” is richer than static HTML, but less so than Ajax. It involves enhancing a standard form with dynamic behavior, so as to make things clearer and help prevent the frustrating validation errors that often come back from the server. For instance, DHTML can be used to ensure a user enters only digits into a credit card field or to add some pop-up instructions for a form field.

Related Patterns

Display Morphing, Page Rearrangement

Display manipulation, as discussed in *Display Morphing* and *Page Rearrangement* (Chapter 5), is often triggered by User Events.

XMLHttpRequest Call, IFrame Call

Web remoting, as discussed in *XMLHttpRequest Call* and *IFrame Call* (Chapter 6), is often triggered by User Actions.

Scheduling



Cron, Event, Future, Loop, Periodic, Plan, Repeating, Schedule, Sequence, Timeout

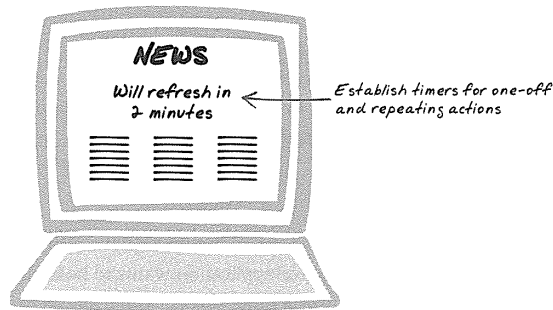


FIGURE 7-3. Scheduling

Goal Story

Frank's panning across a map of the factory. To ensure he monitors all regions, each is color-coded according to how recently it was investigated. It's implemented with Scheduling: after 5 idle minutes, the room turns orange; after 10 minutes, it turns red.

Problem

How can you run actions in the future, or repeatedly?

Forces

- Sometimes, an application needs to repeatedly run the same action; e.g., to extract new data from the server or to monitor application state.
- Sometimes, an application needs to run an action at some future time; e.g., to warn a user his session is about to time out.
- The server can't initiate a connection to the client, so there's no way to have the server "wake up" the client according to a schedule.

Solution

Use JavaScript timers to schedule actions. JavaScript's timer mechanism lets you schedule a one-off action or an action repeating at fixed intervals. In either case, what you specify is an action and a period of time in milliseconds. Note: an online demo (<http://ajaxify.com/run/scheduling>) illustrates the code concepts throughout this section and the code snippets loosely follow from the demo.

The naïve way to run an event in the future would be:

```
sleep(5000); // Nope, won't work.  
expire();
```

That won't work because JavaScript doesn't have a `sleep()` capability—you can't just block in the middle of a script.* Instead, you need to schedule the execution.

The most basic usage is planning a one-off event in the future. For example, suppose an e-commerce application wants to expire a price offer after five seconds:

```
setTimeout(expire, 5000);  
function expire() { $("price").innerHTML = "Expired"; }
```

What if something happens and you want to cancel the timer. For example, the user starts typing in a deal quantity, and the e-commerce application wants to hold off to give the user some more time. `setTimeout` actually returns a timer object, which allows for cancellation:

```
var expiryTimer;  
...  
expiryTimer = setTimeout(expire, 5000);
```

* There are some workarounds (http://www.faqs.com/knowledge_base/view.phtml/aid/1602/fid/143), though they're not really suitable for production.

```

$("dealQuantity").onkeypress = function() { // User typed something.
    clearTimeout(expiryTimer);
};

```

In this example, it might make more sense to *postpone*, rather than cancel, expiry altogether. You can achieve this by creating a new timer:

```

var expiryTimer;
...
expiryTimer = setTimeout(expire, 5000);
$("dealQuantity").onkeypress = function() { // User typed something.
    clearTimeout(expiryTimer);
    expiryTimer = setTimeout(expiryTimer, 2000); // 2 secs more after a keystroke.
};

```

So far, the future action has been a single function call (`expire()`). Sometimes, it's more convenient to say what happens as part of the timeout, in which case you can wrap it all in a string. This prevents the need to create a function specifically to handle the timeout. The string will be evaluated upon timeout:

```

setTimeout("'" + $('price').innerHTML + 'Expired"', 5000); // Got rid of the function.

```

A string is also useful when you want to specify an argument, either fixed or dependent on some variable:

```

setTimeout("expireWithMessage('The deal is off!')", 5000);
setTimeout("expireWithMessage(name + ' deal is off!')", 5000);

```

You can pass a function instead of a string:

```

setTimeout(function() {
    expireWithMessage(name + ' deal is off!'); // Caution!
}, 5000);

```

That will work, but beware: the expression will evaluate at time of execution, not declaration. The `name` variable will resolve to the value of `name` when the timer fires, not when it's created. To make it more concrete, here's a script that issues two alerts. What do you think they say?

```

var name = "Ye Olde DotCom";
setTimeout("alert('Dealing with ' + name + '')", 3000);
setTimeout(function() {
    alert(name + ' deal is off!'); // Caution!
}, 5000);
name = "New Formula 2.0";

```

If you run this script at <http://ajaxify.com/run/scheduling/name/>, you'll see two alerts:

```

Dealing with Ye Olde DotCom
New Formula 2.0 is off

```

What's going on? In the first case, `name` is packed into a new string object when the timer's created. That complete string will be used when the timer fires. In contrast, the second case involves a variable, `name`, which—being an ordinary JavaScript variable—is a pointer

to some memory location. The value in the memory location will only be looked up when the timer fires.

How, then, do you pass an argument to the scheduled function so that it will evaluate when you set up the timer rather than when the timer actually fires? The easiest way is to build up a string, as shown above, but that's ugly for a long block of code. There's another technique based on closures, illustrated in a further refactoring (<http://ajaxify.com/run/scheduling/name/>). It's based on an idea by Scott Isaacs; see his explanation for more details (<http://spaces.msn.com/members/siteexperts/Blog/cns!1pNcL8JwTfkkjv4gg6LkVCpw!340.entry>).

The second type of Scheduling is repetition, and the mechanism is almost identical to one-off event handling. Instead of `setTimeout`, use `setInterval`. Again, the call will return a timer object, useful if you want the option of canceling the loop with `clearTimeout`. The second argument is again a period of time, but with `setInterval`, it represents the loop interval. The following code will call `refreshData` every five seconds:

```
setInterval(refreshData, 5000);
```

A common alternative is to loop with `setInterval`. Here, a new call to the same function is rescheduled, usually subject to some condition. Thus, the function call is repeated until some criterion has been reached.

The timing mechanism isn't super-precise, especially when the user's running lots of programs at the same time and the browser's managing several web pages at once. If timing is important, be sure to test on the targeted browser platforms and ensure your program compensates for any lag. For instance, you might need to cut off an animation effect if periodic timestamp queries suggest it's taking too long.

Real-World Examples

Claude Hussenet's Portal

Claude Hussenet's portal (<http://claudehussenet.com/>; see Figure 7-4) shows various news information, and uses a timer to keep it fresh, as discussed in *Periodic Refresh* (Chapter 10).

Google Suggest

Google Suggest (<http://www.google.com/webhp?complete=1&hl=en>) offers *Suggestions* from the server as you type a query (Figure 7-5). Instead of submitting the query string upon each keystroke, it uses a timer to limit how many queries are sent per second. The pattern's described in *Submission Throttling* (Chapter 10).

Apple iTunes counter

As iTunes Music Store neared its 500 millionth song download, Apple decorated its home-page (<http://apple.com>) with a counter that appeared to show the number of downloads in real-time. In reality, it was a *Guesstimate*. Every few minutes, it would grab the real sales data from the server and estimate how many songs are being sold per second. Between

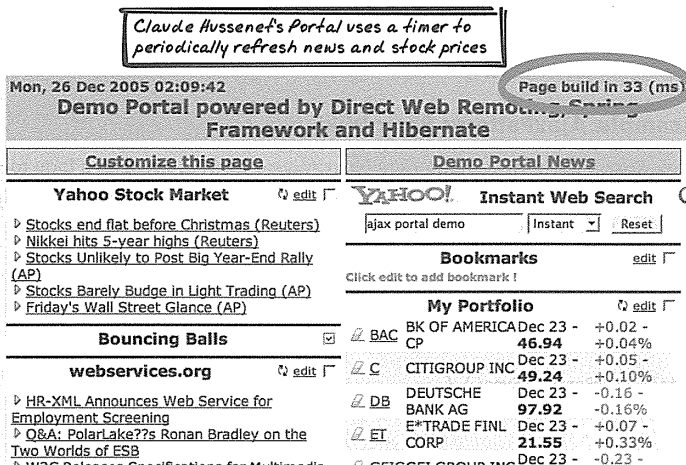


FIGURE 7-4. Claude Hussenet's portal

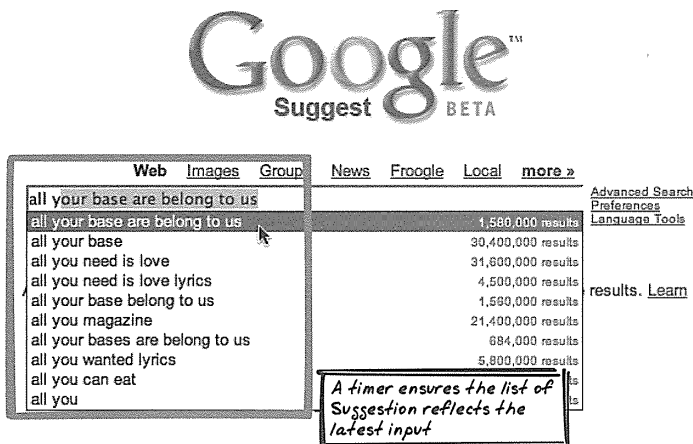


FIGURE 7-5. Google Suggest

those checkpoints, it would use a timer, combined with the estimated sales rate, to continuously update the counter display.

Backpack

37Signals' Backpack (<http://www.backpackit.com/>) maintains items in a list. When data changes, it uses a visual effect known as the "Yellow Fade Technique," where some text

lights up and then fades away. As with most visual effects (see *One-Second Spotlight*, *One-Second Mutation*, and *One-Second Motion* [Chapter 16]), there's a reliance on timers to coordinate the display across time.

Code Example: AjaxPatterns Basic Wiki

The *Periodic Refresh Basic Wiki Demo* (<http://ajaxify.com/run/wiki>) involves a loop to synchronize with the server. The functions to start and stop the loop are encapsulated in their own functions, so as to hide timer details from the rest of the code:

```
function startPeriodicSync() {
    stopPeriodicSync();
    syncTimer = setInterval(synchronise, 5000);
}

function stopPeriodicSync() {
    clearInterval(syncTimer);
}
```

How are these used? Upon loading, an initial synchronization is performed, and `startPeriodicSync()` is called to synchronize thereafter. When the user starts typing inside a message, `stopPeriodicSync` is called, and the loop starts up again when the focus leaves the message area:

```
window.onload = function() {
    synchronise();
    startPeriodicSync();
}

function onMessageFocus(event) {
    ...
    stopPeriodicSync();
}

function onMessageBlur(event) {
    ...
    startPeriodicSync();
}
```

Alternatives

HTTP Meta Refresh

The HTTP Meta Refresh tag schedules the browser to load an entire page at some future time. It's used by conventional news portals, for example, to refresh all content every 15 minutes or so. However, it's very limited since it accepts no input from the browser and forces a complete page refresh, thus destroying all browser application states.

Metaphor

Alarm clocks perform an action at a specified time, and most also have a repetition capability allowing them to annoy their owners at the same time every day.

Extended Technologies

PARADOXICALLY, WE ROUND OUT THE FOUNDATIONAL AJAX TECHNOLOGIES WITH A PATTERN THAT'S all about “non-Ajax” technologies. The term “Ajax” usually relates to the technologies provided by standard web browsers—those covered in the previous chapters. However, it's pragmatic to bend that definition when there are useful “bonus” features that require extra capabilities; hence, the *Richer Plugin* pattern.

Richer Plugin



ActiveX, Applet, Extension, Flash, Greasemonkey, Hack, Java, Mashup, Plugin, Progressive-Enhancement, Remix

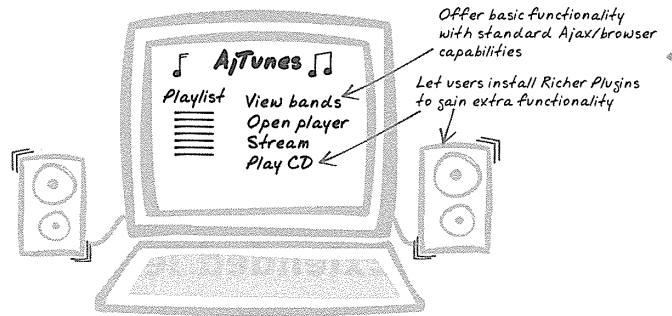


FIGURE 8-1. Richer Plugin

Goal Story

Sasha is using an online photo-sharing web site. Since she's become a big fan, she's taking the time to install some optional plugins from the site: an in-browser photo editor, a screen capture tool, and a browser toolbar.

Problem

How can you make an Ajax App even richer?

Forces

- Ajax makes web interaction much more like a desktop application, but there are limits. Many applications require functionality not available to Ajax Apps.
- If you develop a desktop application, even if it connects to the Internet, you forego many advantages of an Ajax App.

Solution

Make your application “more Ajax than Ajax” with a Richer Plugin. Ajax is certainly a step up in richness, but sometimes you need more than Ajax can offer. That's when you release a Richer Plugin that users install to get even more out of your application. The concept here is “progressive enhancement”—the application works fine in standard web browsers, but by installing the Richer Plugin, users can take the interaction to the next level.

What are the things that a Richer Plugin can achieve that an Ajax App can't? Here's a summary:

Browser morphing

Adding buttons, toolbars, bookmarks, icons; changing browser behavior.

Local file access

Reading and writing files on the user's hard drive; unlike cookies, we sometimes want very large storage capacity and the ability to keep the data locally, without ever transferring to the server.

Sound

Playing music and sound effects.

Rich graphics

Providing rich graphics, changing dynamically; although this is changing with the gradual introduction of technologies like scalable vector graphics (SVG), even these can't come close to utilizing the power of a modern display.

Hardware access

Providing input from devices such as microphones, webcams, and gamepads; output to devices like printers and portable gadgets.

Extended communication

Providing communication from the client machine to locations beyond just the base server, and in protocols other than plain old HTTP.

Operating system interaction

Catching events such as shutdown initiation, changing preferences, popping up alerts, and reading hardware information.

Keyboard shortcuts

Providing a full range of keyboard shortcuts, including those that are platform-specific, while avoiding conflicts with the browser's own keyboard shortcuts.

The browser security model imposes these restrictions on web sites because it lets users freely surf the Web with no risk of being compromised (well, low risk). In its absence, users' systems would be in constant danger from malicious web site owners, complacent web site programmers, and devious users. By releasing a Richer Plugin, you allow users to say, "I trust this site to do certain things with my computer or browser," while keeping all other web sites tied down as before.

This pattern uses a loose definition of Richer Plugin to mean anything outside standard web technologies, which can take several forms:

- Hooking into an existing browser plugin framework, like Flash, Java, or Real. These are third-party plugins that are very common in many browsers. They're often packaged with the browser, and even if not, there's a good chance the user will have previously installed them for another application. Your application should work automatically as long as the plugin's installed.
- Building a browser-specific plugin. Users will then have to explicitly install it.

- Building a companion desktop application. This is not a replacement, because the user primarily interacts through the web site, but a means for better desktop integration, with features like pop-up alerts or quick entries. Many plugins of this nature integrate with frameworks such as Konfabulator (<http://konfabulator.com>) and Growl (<http://growl.info/>).
- Exploiting browser-specific functionality, such as the scalable vector graphics (SVG) capability introduced in Firefox 1.5 or IE's ActiveX components.

The main browsers all have a standard “plugin” mechanism, often used for standard third-party platforms like Java and Flash. There are simpler ways to extend behavior if you are building your own plugin. In IE, you can include ActiveX Controls, and Firefox has its own extension framework* (<http://developer.mozilla.org/en/docs/Extensions>).

A Richer Plugin should be implemented with caution. It breaks browser portability, one of the great advantages of Ajax Apps. It creates a layer of complexity to the web site, which must target users who do and don't have the plugin installed. It also demands a different skill set from standard web development. As a rule of thumb, use Richer Plugins only when there's no standard way to implement the same functionality.

Decisions

Will you reuse a third-party plugin or develop your own?

Having decided that you need extra functionality, the biggest decision is whether or not you'll to build a browser-specific plugin or to build on a standard plugin platform like Java or Flash.

A standard plugin platform has the following advantages:

- The plugin will likely save a lot of development if there's already a specialized API.
- The plugin is usually portable across browser versions. If you write a program in Java, for example, it should run the same no matter which browser and operating system it's used on.
- The plugin might already be installed on the user's browser. Often, browsers come with Java and other plugins preinstalled. And if not, there's a good chance the user's previously installed them.
- The user—or his employer—might trust it more than your own system. People generally trust brands like Flash and Java, so they might install those plugins and nothing else.

* This pattern doesn't discuss the Greasemonkey Firefox extension because, although a great tool, its primary purpose is to let users re-purpose an existing web site, and just about anything that can be performed with Greasemonkey could also be performed by the web site owner.

However, there are also advantages of writing your own plugin:

- You're not locked in to a relationship with the plugin provider. You're therefore in control of issues such as licensing and management of user data, and not at the mercy of another company's future plans for the plugin—or lack thereof.
- You can implement more functionality. Plugin platforms impose their own security model because it's expected different applications will run on them. So while Flash, for example, lets you extend the browser's multimedia capabilities, you can't use it to morph the Firefox toolbar.
- You can target browser versions not supported by a particular plugin platform.
- You can control the plugin installation process, as opposed to sending the user over to a third-party site.

What will happen if the plugin's not installed?

You'll need to consider the user experience when the plugin's not installed. How will you treat content—and links to content—that requires the plugin? For example, you could hide it altogether, explain what would be present if the plugin is installed, or perhaps provide an alternative representation. You also need to consider what to say if the plugin's not available on the user's platform.

In extreme cases, you could make the plugin mandatory. This might be plausible in an intranet setting and is really quite similar to a standard desktop application. However, it might be preferred to a desktop application if users are more comfortable in a web environment.

Real-World Examples

Amazon Mini Shop

Amazon Mini Shop (<http://extensionroom.mozdev.org/more-info/amazonsearch>) is a Firefox extension that lets users search within the sidebar and summarizes the results (Figure 8-2).

Google Toolbar and Google Suggest for Firefox

Google makes available a toolbar (<http://toolbar.google.com>) for both IE and Firefox. The toolbar augments standard browser behavior with a convenient search entry as well as several other features, including Google's page rank metric for the page. Google openly notes that they may collect information about web pages being viewed, data that many people assume is used to help fine-tune their search algorithms.

Google also offers a "Google Suggest for Firefox" extension (<http://toolbar.google.com/firefox/extensions/suggest/install.html>). This is a toolbar version of the Ajax Google Suggest web site (<http://www.google.com/webhp?complete=1&hl=en>).



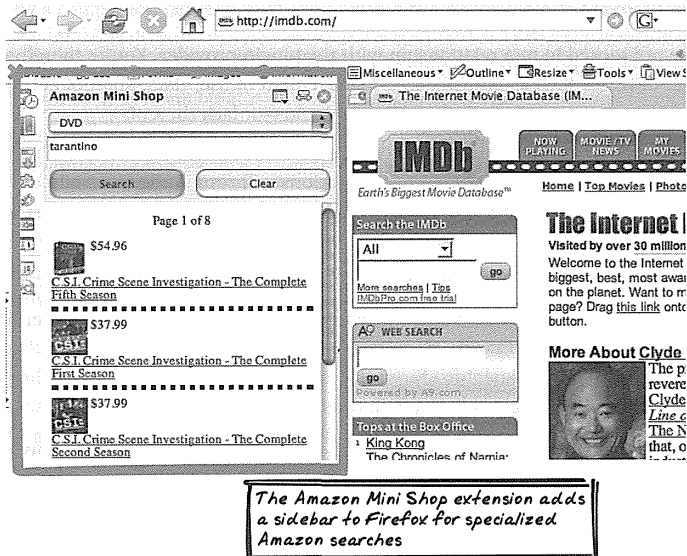


FIGURE 8-2. Amazon Mini Shop

Odeo

Odeo (<http://odeo.com>) is a podcast manager (Figure 8-3). The main web site interaction— functionality like subscribing and tagging—uses standard web technologies, with many Ajax techniques. In addition, Odeo offers a number of Richer Plugins. Flash is used to play and record podcasts in the browser. For pulling down podcasts to a local machine, there's an Odeo "Syncr" desktop tool and also a more specialized Apple Dashboard widget.

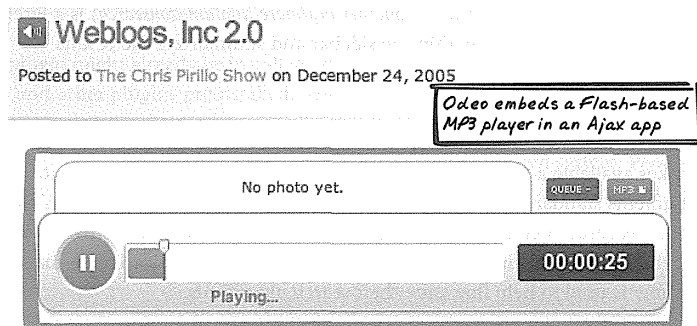


FIGURE 8-3. Odeo

TiddlyWiki

Jeremy Ruston's TiddlyWiki (<http://tiddlywiki.com>) is a SPA application (defined in the section "Alternatives," later in this chapter), an interesting variant of Richer Plugin. It's an entire wiki within a single HTML file, including the wiki script and the wiki content itself. Tiddlywiki is similar to Richer Plugin because it overcomes the local storage barrier by requiring users to save the wiki page locally and point the browser to that file.

Code Example: Amazon Mini Shop

It's beyond the scope of this book to cover plugins in detail, but this example gives an insight into the components of a Firefox extension, Paul Millar's Amazon Mini Shop Firefox extension (<http://extensionroom.mozdev.org/more-info/amazonsearch>). If you want to grab the code, visit the link with Firefox and install the extension. Then, look at *Extensions.rdf* in your firefox profile directory (e.g., under `~/.mozilla/firefox/` in Unix or under `c:\Windows\Profiles\%USER%\Mozilla\Firefox`). Search for "amazoncomsearch" and you should see a nearby tag beginning with `RDF:Description`. Within that tag, the extension ID is in curly braces, and you need to go into a directory by that name, which is where the extension code is.

The extension contains several components. To start with, *chrome/amazoncomsearch.jar* is a jar file that can be opened with a standard unzipping utility. It contains the main resources for the extension. In Firefox, user-interfaces are set up with the XML User Language (XUL) (<http://xulplanet.com/tutorials/xultu/intro.html>), a format that looks similar to HTML. The sidebar UI is captured in one XUL file:

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<page id="pageAmazonPanel" style="background-color: #ffffff;" title=
"Amazon Search"
    xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul" xmlns:
html=
"http://www.w3.org/1999/xhtml" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
onload=
"disable();javascript:document.getElementById('txtAmazonSearch').focus()">

    <script type="application/x-javascript" src="chrome://amazoncomsearch/content/
amazonsearch.js"/>
    <hbox pack="center">
        <menulist flex="1" id="menuAmazonSearch" label="Search Category" onChange=
"change()">
            <menupopup id="popupAmazonSearch" onpopupshowing="popamazon()"/>
        </menulist>
    </hbox>
    ...
</page>
```

There is also JavaScript code, which populates the UI and handles user-interface interaction similarly to a normal browser script. For example, an *XMLHttpRequest Call* is used to gather data from Amazon's web service:

```
aURL = "http://xml- "+value+".amazon.com/onca/xml?Service=AWSECommerceService&..."
...
httpReq.open("GET", aURL, true);
httpReq.send(null);
httpReq.onload = httploaded;
```

Among the remaining components, *chrome.manifest* is a meta-information file that points to the resources above. *install.rdf* provides meta-information about the application, such as its ID and creator. *install.js* is a script to handle initial installation of the extension. *defaults/preferences/ams.jar* provides defaults that are stored as standard Firefox preferences. In this case, there is only one preference: the Amazon site being searched (U.S., U.K., etc.). (You can see the settings in *prefs.js*.)

Alternatives

Desktop Client

The desktop client runs as a standard application in the user's operating system and connects to one or more servers using HTTP or any other protocol. It would make sense to use a *Desktop Client* if the required user interface is very different than a standard browser interface, or if it's difficult to extend the browser as desired. While the plugin frameworks provide more control than regular web applications have, the developer is still restricted relative to a standalone desktop client.

A variant is a "Richer Desktop Client." That is, an Ajax App to do most basic things, and a richer desktop client with extra features and improved usability. This is the strategy Microsoft might follow with Office Live.

Related Patterns

Fat Client

Fat Clients (Chapter 13) are sometimes replacements for desktop applications, used frequently and for long sessions. A Richer Plugin will enhance their experience, and the effort to set it up will eventually provide users with ample payback.

Single Page Application

Single Page Application (SPA) is a newer-than-Ajax term that describes an application whose entire logic and data are embedded in a single HTML file. Point the browser to the HTML file on a local drive; editing the data will update the file accordingly. In the extended case of Single Page Application and Development Environment (SPADE) (<http://trimpath.com/project/wiki/SinglePageApplicationAndDevelopmentEnvironment>), you can even update the code via the page itself! SPA is useful for certain specialized local applications where you want a web interface and local storage. A nice way to use it is to load it onto a

USB key along with a web browser, which allows you to maintain the content anywhere, even in the absence of a Net connection.

Bookmarklet

A bookmarklet (or “favelet”) is a link that references JavaScript instead of a web location. For example, you can create a link like this: `<a href="javascript:alert('You are at' + document.location.href + '.');">Bookmark Me!</a>`. The user will be able to drag it to their toolbar, and when they click on it later on, it will run the script, which pops up the current location in this example. With a bookmarklet, you can build a custom Ajax App inside other web sites. Display Manipulation (Chapter 5) patterns and Dynamic Behavior (Chapter 7) patterns work as on a local site, so you can, for example, tweak any web site whenever the user clicks on the bookmarklet. One thing you can't do is issue *XMLHttpRequest Calls* (Chapter 6), due to the same-origin policy, but you can work around that with *On-Demand JavaScript* (Chapter 6). *WizLite* (<http://wizlite.com>), a collaborative annotation tool, is a great example of how powerful bookmarklets can be.

Browser-Side Cache

A Richer Plugin, with access to the local filesystem—or at least more sophisticated browser-based storage than normal cookies—can be a useful data repository for a *Browser-Side Cache* (Chapter 13).

Metaphor

Consider a regular web app as a black-and-white movie. Ajax adds color; Richer Plugin adds 3-D (or Smell-O-Rama if you wish).

Want to Know More?

- Firefox Extension Tutorial (http://extensions.roachfiend.com/howto_bug.html)
- Building ActiveX Controls for Internet Explorer (<http://msdn.microsoft.com/workshop/components/activex/buildax.asp>)
- Tutorial: Creating a Mozilla Extension (<http://www.mozilla.org/docs/tutorials/tinderstatus/>)

Programming Patterns

KNOWING THE FOUNDATIONAL TECHNOLOGIES IS ONE THING; APPLYING THEM IS ANOTHER. AJAX IS NOT rocket science, but there are still plenty of technical issues that will arise in building a serious Ajax App. The programming patterns focus on technical qualities of software—in particular:

Maintainability

Ensuring web services and browser code are easy to understand and work with

Robustness

Ensuring the application will prevent errors where possible and handle them gracefully when they do arise

Performance

Ensuring the application runs as quickly as possible and feels smooth and responsive from the user's perspective

The first chapter in this section, Web Services (Chapter 9), provides several alternative strategies for designing web services. Also related to Web Remoting, Browser-Server Dialogue (Chapter 10) looks at the flow of information between browser and server; e.g., synchronization techniques.

DOM Population (Chapter 11) covers several strategies for handling DOM population following a server response. Code Generation and Reuse (Chapter 12) contains a couple of general programming patterns for maintainability and portability. Finally, the Performance Optimization (Chapter 13) patterns are about improving not only speed of updates and communication, but also about optimizing the user experience in the face of inevitable delays.

Web Services

THE CONCEPT OF A WEB SERVICE HAS ALREADY BEEN INTRODUCED IN THE *WEB SERVICE* PATTERN, BUT the pattern leaves open many questions—for instance, what sort of messages will be sent to and from the service? How will the URLs look? How will the service relate to backend software? These are the kind of design and programming issues covered in this chapter.

RESTful Service is an overview of Representational State Transfer (REST), a popular set of patterns and idioms for consistent and intuitive web services. *RPC Service* has similar goals, but with a philosophy that leads to very different services. Neither REST nor Remote Procedural Call (RPC) are Ajax-specific, because they are general industry conventions for web services. However, the next pattern, *Ajax Stub*, is Ajax-specific, because it provides a convenient way to invoke RPC-style services from the browser.

The remaining patterns look at popular message formats used for communication between the browser and *Web Services*. *HTML Message* is a message in HTML form, usually sent from the server to be morphed directly onto the page. In contrast, the remaining messages are raw responses that usually can't be displayed directly, and that may be transferred in both directions. They are *Plain-Text Message*, *XML Message*, and *JSON Message*.

RESTful Service



API, HTTP, REST, Standard, Universal, Web

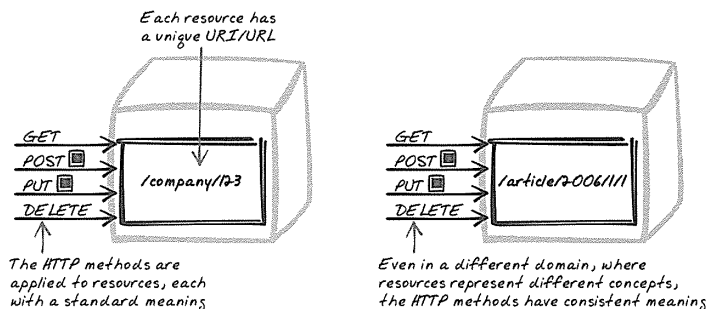


FIGURE 9-1. RESTful Service

Developer Story

Devi's producing a public web site explaining pharmaceuticals to the public. The browser applications will pull down drug data from the server, and authorized users—such as drug manufacturers—will be allowed to upload new information. It's envisioned that third-party applications will also connect to the web server and provide their own user interfaces. For example, pharmacies will be able to use the information via their point-of-sale systems, and doctors will have access from their specialized consulting software. Devi opts to follow a RESTful architecture for all of the services exposed on the server. This makes it easy for the browser side of her own web app to access the data and exposes services in a standardized manner.

Problem

How do you expose services on the Web?

Forces

- The browser side of an Ajax App needs to call server-side services.
- There might be more than one browser-side application sharing the same server-side service.
- As well, it's often desirable for third-party applications to invoke the server-side service too.
- With numerous different applications—and a variety of developers—accessing the service, it ought to be easy to use.

- Taken as a whole, the server-based services in a single “application” form an Application Programming Interface (API). It’s important that services work consistently within the API.
- The basic architecture of the Web does not force any type of service architecture—any given functionality could be exposed in a wide variety of styles.

Solution

Expose web services according to RESTful principles. Representational State Transfer (REST) is an architectural style—or “pattern”—guiding the architecture of web services. Like “Ajax” itself, “REST” is a broad term and can be broken down into many smaller patterns and idioms (enough to fill up an entire book!). The explanation here can’t possibly do justice to the entire concept, but it’s worthwhile being familiar with the general principles of REST. Because REST is such a large topic, be prepared for the reality that most real-world systems you’ll encounter are only *somewhat* RESTful; they’ll follow some principles and break others. Some people are now making the distinction between “High REST”—sticking closely to REST—and “Low REST”—following just a couple of core REST guidelines (<http://lesscode.org/2006/03/19/high-low-rest/>).

Motivating REST: many ways to skin a call

To motivate REST, let’s consider a football service. Sufficiently authorized clients can read game records, upload new games, correct existing games, or delete games. We’ll assume your backend’s already written, but how would you offer it as a web service? How will client developers call it, and how will they learn about its interface?

To wit, following are a few random ways you might expose a game service—*note that these aren’t necessarily RESTful!*

- GET call to a special newGame service, `http://example.com/newGame?id=995&bluebaggers=150&redlegs=60`
- POST or PUT call to a special newGame service, `http://example.com/newGame`, with an XML body like `<game id="995"><score team="bluebaggers">150</score><score team="redlegs">60</score></game>`
- POST or PUT call to a special newGame service, `http://example.com/newGame`, with a CGI-style body: `id=995&bluebaggers=150&redlegs=60`
- GET call to a multipurpose gameMaintenance service, `http://example.com/gameMaintenance/command=newGame&id=995&bluebaggers=150&redlegs=60`
- POST call to a multipurpose gameMaintenance service with a command-qualified URL, `http://example.com/gameMaintenance/command=newGame`, with an XML body like `<game id="995"><score team="bluebaggers">150</score><score team="redlegs">60</score></game>`

And we could go on. And then we could look at the other functions here. The point is this: as a service designer, you have many options for representing a single function. Is that, in itself, a problem? You might argue it’s not a problem at all—just choose any old option,

using some darts and a blindfold if need be. Then just create the service and document its usage—any competent developer will then be able to deal with it, right? That argument, however, is flawed on several counts:

- You might assume that the browser script for your web app, and perhaps some known external clients, are the only users of your service, making it relatively easy to create and document any service. In fact, your application resides within a larger Internet ecosystem. Among the entities involved: the web browser (e.g., Firefox), which impacts issues like back-button handling and bookmarking; the web server (e.g., Apache), which impacts issues like caching and compilation; network routers and caches all along the way; robot applications such as those crawling the Web on behalf of search engines; and personal web agents that trawl through the Web on behalf of individuals. Your application will perform faster and more reliably if you follow certain conventions that are common among these entities.
- Ideally, a service should be intuitive and self-documenting: if a service is based on familiar conventions, you have only to specify the bare minimum for a developer to be able to use it. That's simply not possible if everyone chooses alternatives according to their own personal taste. The standards-based interface usually trumps the optimized interface that's technically superior but based on unfamiliar conventions. There are thousands of web services available on the Internet; developers should be able to assess them and put them into action with as little effort as possible.
- In the absence of any guiding principles, there's a disturbingly high likelihood you'll end up in tedious debates with colleagues over the most trivial issues, trying to arrive at an optimal API. (The blindfold and darts—or an impatient manager—would alleviate this kind of paralysis, but the other issues would still remain.)

So we've seen the argument for a convention-based approach to web services, but what's the convention? REST is one such convention, consisting of many different guidelines, that has broad industry support. Just about any functionality can be presented as a RESTful API, and by doing so, it gains the advantage of a familiar, intuitive style. Many services on the Net already follow RESTful principles, to differing degrees. Also, many real-world entities such as browsers and caches tend to assume REST. These entities work better if REST is in place—a browser, for instance, might give better feedback; a cache will retain all the data that the programmer expected it to retain. And what will happen if REST is not in place? It's probably not the end of the world, but you'll find little things will go wrong and work in unexpected ways. For instance, hitting the browser's Reload button might cause a purchase order to be resubmitted. Or a cache might end up caching data it shouldn't, or it might not cache any data at all. Why these unfortunate events might occur will become clear as we learn about the RESTful principles.

Introduction to RESTful principles

So much for abstract motivation: what exactly are the RESTful principles (<http://www.xfront.com/REST-Web-Services.html>) all about? REST ultimately presents the server as a big blob of “resources” such as people, cars, or football games. Clients interact by inspecting

and changing that state. Putting a purchase order on the server, for example, will trigger the server to affect the purchase described in the order. This contrasts with the Remote Procedural Call (RPC) approach, where the client would call a remote `purchase()` procedure. REST is based on data manipulation, RPC is based on procedural calls.

REST sees the universe as consisting of resources and operations performed on, and with, those resources. The resources and the operations can be represented as the HTTP concepts of URLs and call methods.

Resources as URLs

A resource is something like a “business entity” in modelling lingo. It’s an entity you wish to expose as part of an API. Almost always, it’s a noun; e.g., a person, a car, or a football game. Each resource is represented with a unique URL. So when you say a RESTful URL, it will be a thing rather than an action.

Operations as HTTP methods

REST leverages the existing HTTP methods, particularly GET, POST, PUT, and DELETE. Note that the `XMLHttpRequest` object, and some wrapper libraries such as `ajaxCaller` (<http://ajaxify.com/run/testAjaxCaller/>), support all these methods. If they’re unfamiliar to you, consult the W3C HTTP 1.1 Specification (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>).

Another way to say it is in HTTP, any request a client can make involves a URL and an HTTP method. With REST, the URL is designed to represent a noun and the HTTP method always maps to one of several standard verbs, which will be performed against that noun.

How is the URL combined with the HTTP methods in practice? Returning to the example, a game is clearly an entity in this API. So REST says we create a scheme to expose each particular game as a unique URL—for example, <http://example.com/games/995>.

So far, this is all pretty unequivocal. Anyone familiar with REST will always be driven to create a unique URL for each game. There might be differences of opinion about the precise naming, but the basic concept will always be the same if you follow REST. Now that we have this unique URL, we can implement many functions around it, by leveraging the standard HTTP methods (<http://rest.blueoxen.net/cgi-bin/wiki.pl?RestFaq#nid1UP>).

- To get game results, the client simply issues a GET call on the URL. GET is the standard way to perform queries on the resource indicated by the URL. The response format is not dictated by REST, but XML is common due to its self-describing and self-validating nature.
- To delete the game, the client issues a DELETE call on the URL. DELETE is the standard way to delete a resource.
- To update a game, the client builds a fresh game message and uploads it as the body of a PUT command. PUT is the standard way to update a resource’s value.
- To create a new game, the correct action will depend on whether the client can guess what the game’s URL will be. If the service just allocates the URL arbitrarily, perhaps

URLS OR URIS?

In strict HTTP parlance, there's a distinction to be made between Universal Resource Locators (URLs) and Universal Resource Identifier (URIs). URI is the more general term, as it describes a general resource. A URL is a particular type of URI that, in a somewhat vague sense, identifies a network location. You could define a protocol to represent books with URIs (e.g., `isbn://12345`), and those URIs wouldn't be URLs since they have no relationship with any network location; they are just a scheme for uniquely identifying objects. A string such as `http://ajaxpatterns.org/RESTful_Service` not only identifies an object (making it a URI), but also refers to a network location (making it a URL), one which will be used by various software to locate the resource.

Having done the Right Thing by pointing out the distinction, let me state why this pattern refers only to URLs. Most importantly, URLs are the language most developers are familiar with, even in contexts where URI is technically the appropriate term (which is arguably most of the time). Furthermore, there is confusion (<http://kyseo.blogspot.com/2005/12/uri-url-urns.html>) even among the RFCs that define these terms and the definitions have also shifted over time. The W3C states:

People who are well-versed in URI matters tend to use "URL" and "URI" in ways that seem to be interchangeable. Among these experts, this isn't a problem. But among the Internet community at large, it is. People are not convinced that URI and URL mean the same thing, in documents where they (apparently) do. When one sees an RFC that talks about URI schemes (e.g., [RFC 2396]), another that talks about URL schemes (e.g., [RFC 2717]), and yet another that talks of URN schemes ([RFC 2276]) it is natural to wonder what's the difference, and how they relate to one another. While RFC 2396 1.2 attempts to address the distinction between URIs, URLs and URNs, it has not been successful in clearing up the confusion.

Given the potential for confusion, and the fact that most developers exclusively use the term "URL," I've opted for the pragmatic approach: you'll only see "URL" used throughout this pattern, and indeed, throughout all others.

based on an auto-incrementing counter, then the client won't be able to guess it. When that happens, the new game is POSTed to a URL like `http://example.com/games`, and the server responds with the new game's URL (`http://example.com/games/995`). On the other hand, the client might be able to guess the URL, based on a naming convention. If the client guesses that the new game would have URL `http://example.com/games/2000/9/25`, then it can simply PUT the new game to that URL.

- To perform an arbitrary transaction, you'll also use one or more of these standard methods. So maybe the client wants a way to double the winner's score. In this case, there's probably no special service—the client just reads the old match, updates it, and

PUTs it back to the server. For more complicated transactions, you might design a new resource specifically to encapsulate the request (e.g., a “purchase order” resource), and the client will then be able to POST resources like this to the server. Many businesses work the same way—contracts/orders are passed around in order to request that things are done and to verify that they’ve been done. See the section “Handling arbitrary transactions” later in this chapter for more details.

In summary, each HTTP method will cause a well-defined action on the resource represented by the URL it operates on. The methods can be compared to SQL commands: GET is like “SELECT,” DELETE is like “DELETE,” POST is like “INSERT” with an auto-generated ID, and PUT is like “INSERT OR UPDATE IF EXISTS” with an ID specified.

Again, notice how most of this is unequivocal. True, you still have to decide on a precise URL convention and message formats, but that’s fairly minimal. *Just by declaring “this is a REST interface,” you’re conveying many implicit rules to a REST-savvy developer.* Inform that developer of the URL and message conventions, and they’ll be able to intuit most of the API. Compare that to an ad hoc, roll-your-own API, where each detail must be explained piecemeal. Considering that there are thousands of web services out there, it soon becomes clear why REST is an attractive option.

RESTful principles

Let’s now look at the main REST principles in more detail.

URLs reflect resources

As explained in the previous section, each resource receives a unique URL. URLs generally look like nouns.

REST doesn’t tell you what names to call your resources—there are infinite possibilities and the choice is up to you. They’ll often reflect your conceptual model of the system and look similar to your database table names.

HTTP methods reflect actions

With REST, each HTTP method indicates the kind of action to be performed. Moreover, the REST API designer doesn’t have to decide what each method means, because there are only a handful of methods, and REST standardizes the meaning of each—GETs for queries, POSTs for inserts, and so on, as explained above. Notice how resource names are *not* standardized in the same way. Businesses have different models, so you can’t standardize on those things. But REST is saying that you can indeed model the resources so that the set of required actions is standard, thus ensuring APIs are consistent with one another.

Fortunately, the standard HTTP methods—when combined with a well-designed URL and resource scheme—are enough to let a client express just about anything it needs to. It’s true that in an ideal world, there would be a few extra actions to make life a bit easier. For example, it would be convenient if there was a standard *LOCK* action to let a client deal

exclusively with a particular resource.* However, you can still work around this by introducing a lock resource and letting clients lock things by manipulating those resources with the standard HTTP methods.

GET for queries, and only for queries

Refining the previous point, GET calls must be read-only. When you perform a read-only query of any kind, always use GET. Conversely, when your call will affect any data on the server, use one of the other methods: PUT, POST, and DELETE all change state. There are some caveats here—for example, if issuing a query ends up leaving a log message somewhere, does that constitute a change? Technically, yes; but common sense says no, it's not a significant change to the underlying resource, so it's acceptable to log the message in response to a GET request. Just be aware that the log might not read as intended; for instance, some clients will get the content from a cache, which means no log message will occur.

The reverse problem can happen too. GET requests are often cached, but the other types of requests are not. When you perform a query with POST, you deny the possibility of caching in the browser, within the client's network, or on your own server.

If you learn nothing else about REST, be sure to learn this guideline; of all the REST conventions, this is the best known and widely applied across the net (see the sidebar “The Backpack-Accelerator Incident” in this chapter).

Services should be stateless

In stateful interaction, the server remembers stuff from the past. For example, you could do this:

Greg's Browser: Give me the next game to work on.

Server: Work on game 995.

...

Greg's Browser: Here's the result: bluebaggers 150, redlegs 60.

Server: Thanks.

(Server stores “bluebaggers 150, redlegs 60” for game 995.)

The server here has an implicit conversational state; it remembers the game Greg's working on. That makes testing harder, because you have to set the whole history up to test any particular interaction. Another problem is the fragile nature of the transaction; what if Greg's browser fails and ends up requesting two different games? When Greg uploads the score, the server could easily misinterpret which game he's talking about. Also, you can't cache effectively because a query result might depend on something that happened earlier.

* The WebDav protocol does just that. It adds several methods to HTTP, to support locking, property inspection, and moving resources around. These make REST easier to implement, but unfortunately, most Internet entities won't understand them, so it defeats one of the main purposes of REST on the open Internet.

THE BACKPACK-ACCELERATOR INCIDENT

An incident in mid-2005 highlights the importance of following “GET for queries, and only for queries” (http://radar.oreilly.com/archives/2005/05/google_web_acce_1.html). It involved the newly released Google Accelerator interacting with non-RESTful services. The accelerator is a proxy that jumps ahead of the user and prefetches each link in case the user should click on it (a non-Ajax example of *Predictive Fetch* [Chapter 13]). As with many other entities on the Net, it assumes services follow “GET for queries, and only for queries.” The practical implication is that it indiscriminately clicks on any links.

The problem came when users logged into non-RESTful applications like Backpack (<http://backpackit.com/>). Because Backpack deletes items using GET calls, the accelerator—in its eagerness to run each GET query—ended up deleting personal data. This could happen with regular search engine crawlers too, though the issue doesn’t tend to come up because they don’t usually have access to personal accounts.

Some people argued Google went too far with the REST assumption (<http://www.loudthinking.com/arc/000454.html>); it’s one thing to assume REST for caching a prior response, but another to proactively visit links that otherwise wouldn’t have not been touched, especially when those are only visible to a logged-in user. The argument has some merit, but either way, many components will continue to make the same assumption as the Google Accelerator did, making it prudent to practice “GET for queries, and only for queries.”

Furthermore, an objective of REST is to be able to switch clients at any time and receive the same result. If you switch “Greg’s Browser” to “Marcia’s Browser,” the server will respond differently to the game result above, because it will assume Marcia’s Browser is working on a different game number. While users are unlikely to switch browsers mid-conversation, other clients like caches and robots may well switch. Substitutability makes networks more scalable.

The upshot is that everything should be passed in at once. It’s okay for the client to remember details about the conversation—REST says nothing about the client’s activity—but the server must not. The server responses should be based on the server’s global state and not on the state of the conversation; i.e.:

Greg’s Browser: Give me the next game to work on.

Server: Work on game 995.

...

Greg’s Browser: Here’s the result **for Game 995**: bluebaggers 150, redlegs 60.

Server: Thanks, I’ll store “bluebaggers 150, redlegs 60” for Game 995.

So are cookies used at all? Yes, cookies *can* be used, but mainly for authentication. They should only determine *whether or not* the server will accept a call, by not in any way *how* the server will respond. If authentication fails, an appropriate response, such as a 401 (Forbidden) header, will result. If authentication succeeds, the same call will always have the same effect, independent of the client who made the call. So if Greg's Browser says "Get All Games," Greg will see exactly the same thing as if the all-powerful administrator asked for the same thing, as long as Greg's allowed to see them. If the security policy forbids Greg from seeing all games, Greg's query will be denied. What *won't* happen is for Greg to receive just a minimal list of his own games—if that happened, the server would be using the cookie to determine the browser's response. To get a minimal list, the query must be explicit—"Get All of Greg's Games"—and of course, it must be Greg or the administrator that issues this request. If Marcia made that request, she'd get an authentication error.

An alternative to cookies is to use HTTP Basic Authentication, but this doesn't work too well in an Ajax context, because you don't have any control over the UI. The browser will typically pop up a dialog box, whereas you probably want to integrate any authentication directly into the interface, especially if using a pattern like *Lazy Registration*. HTTP Digest Authentication does allow more flexibility, but browser and server support is limited. Thus, while HTTP authentication is seen as "more pure," cookies are a reasonable workaround, provided you use them only for authentication.

Services should be idempotent

"Idempotent" means that once you pass a message to service, there's no additional effect of passing the same message again. Consider the deletion message, "Delete Game 995." You can send it once, or you can send it 10 times in succession, and the world will be the same either way. Likewise, a RESTful GET is always idempotent, because it never affects state. The basic conventions on GET/DELETE/PUT/POST, described earlier, are intended to support idempotency.

Services use hyperlinks

In resource representations returned by GET queries, use hyperlinks liberally to refer to related resources. With judicious use of hyperlinks, you can and should break information down. Instead of providing one massive response, include a modicum of information and point to further information using resource identifiers.

Services document themselves

RESTful services can and should document themselves. The exact mechanism is not well-defined. But a couple of examples include:

- Base URLs can explain how the service works. For example, in the earlier example, <http://example.com/game/995> represents a particular game, but <http://example.com/game> represents no real entity, so its result could be an instructive document, in human-friendly language, with appropriate hyperlinks.

- Error responses should also be in human-friendly language and with examples and hyperlinks.

Services constrain data formats

As an extension to the previous point, RESTful services rely on standards such as Document Type Definitions (DTDs) and XML Schema to verify data formats as well as to document what's acceptable.

Handling arbitrary transactions

The REST definition is especially geared for creating—such as Amazon—reading, updating, and deleting (CRUD) operations. How about transactions and arbitrary actions, such as “Pause the printer” or “Email the catalogue to this user”? By definition, application-specific actions don't fit neatly with the standard REST actions. There will always be some pragmatic judgment required as to a suitably RESTful interface. A few possibilities have been mooted, with viewpoints varying (<http://rest.blueoxygen.net/cgi-bin/wiki.pl?VerbsCanAlsoBeNouns>):

- If it makes sense to do so, reformulate the action as a standard persistence operation. For updates that are complex or entail business logic, you can provide a read-only service—accessed with GET queries—to help the client compute the new state.
- You can POST a message to the resource in question—for example, POST a “pause” message to a URL representing the printer.
- The server can expose a stateless “processor” resource to perform the action—e.g., a printer controller—and have clients POST command-like messages there, with URLs referencing the resources to be acted upon.

Weighing Up REST

Being a broad architectural style, REST will always have different interpretations. The ambiguity is exacerbated by the fact that there aren't nearly enough standard HTTP methods to support common operations. The most typical example is the lack of a search method, meaning that different people will design search in different ways. Given that REST aims to unify service architecture, any ambiguity must be seen as weakening the argument for REST.

Another issue is portability—while GET and POST are standard, you may encounter browsers and servers that don't deal consistently with DELETE, PUT, and other methods.

The main alternative to REST is RPC (see *RPC Service*, later). It's equally broad in definition, but the essential idea is that services are exposed as procedures. You end up POSTing into verb-like URLs such as `/game/createGame?gameId=995` instead of RESTful, noun-like URLs such as `/game/995`. In fact, the distinction is significant enough that some service providers, such as Amazon, actually provide separate APIs for each. As a general rule, any set of services could be exposed as either REST or RPC; it's just a question of API clarity and

ease of implementation. Note also there is some overlap; as discussed in the *RPC Service* solution, RPC can still follow certain RESTful principles.

From an implementation perspective, REST and RPC differ in that REST requires some explicit design, whereas RPC tends to follow directly from the backend software model. In the example, it's likely there will be a *Game* class with a `createGame()` method—that's just how most server-side software gets done. So it's a no-brainer to tack on a `/game/createGame` web service that mediates between the client and the backend method. In fact, there are many frameworks that will completely automate the process for you.

With REST, there's no direct mapping between web service and backend implementation—an impedance mismatch. You need to take a step back and explicitly design your API to be RESTful. If you're following feature-driven design, the API is the first thing you'll produce anyway, since the design will be "pulled" by the needs of clients, rather than "pushed" from the available technology. Once you've designed the API, the web service implementation will effectively be a kind of middleware Adaptor (see Gamma et al., 1995) to the backend services.

To summarize crudely:

- Any web service functionality can be exposed as either REST or RPC. There's a good argument that REST APIs are somewhat clearer, though that view is far from universal.
- REST APIs may be clearer, but they do require more design and maintenance because they tend to diverge from the backend technology. However, adaptor-style implementations of this nature are quite easy, so the overhead is not substantial and often justified in the effort to provide a clean interface, unaffected by the incidentals of the backend implementation.

Real-World Examples

I'm not aware of any Ajax Apps that access a truly RESTful interface on the server side. Consequently, the example here is a public API that conforms closely to REST. Note that several prominent industry interfaces are not covered here, because though they promote themselves as RESTful, they tend to break quite a few basic principles—for example, Amazon's REST API (<http://rest.blueocean.net/cgi-bin/wiki.pl?HowAmazonsRESTComparesWithREST>). In industry parlance, REST is sometimes synonymous with "not SOAP," and it's incorrectly assumed that an interface is RESTful as long as it uses GET for reads and POST for writes.

Blogger API

Atom is a feed protocol built around RESTful principles. Blogger offers a good description on its use of Atom in its public API (<http://code.blogger.com/archives/atom-docs.html>). The API lets third-party applications read and change Blogger blogs. Blogger themselves could theoretically build an Ajax App that directly calls on the API.

A blog entry is one important resource in Blogger's API. For example, entry ID 1000 for user 555 "lives at" <http://blogger.com/atom/555/1000>. So to read that entry, just issue a GET on that URL. To change the entry there, just PUT an XML document to the same URL. To add a new entry, you don't yet know its ID, so you POST to a URL containing only the user ID—for example, <http://blogger.com/atom/555>. Note that each of these operations uses HTTP Basic Authentication and runs over HTTPS.

Code Example: AjaxPatterns RESTful Shop Demo

The Basic Shop Demo (<http://ajaxify.com/run/shop/>) employs an ad hoc web service to expose and manipulate store items and user shopping carts. In this demo, the cart service is refactored to become more RESTful. You can contrast this to the Refactoring Illustration in *RPC Service*. Note that some constraints make the RESTful service less than ideal here:

- URLs would ideally be prefixed with <http://ajaxshop.com>, but because it's running within the Ajax Demo framework, the prefix is longer: <http://ajaxify.com/run/shop/rest>.
- URLs would ideally avoid CGI-style variables and look like `/content/Movies` instead of `/content?category=music`. That's not possible here because there's no use URL-rewriting (to make the demos easy to install).

There are three resources: categories, items, and carts. GET is used to read each of these. Only carts can be modified, either by adding an item or clearing the cart. Both of these are handled by POST rather than PUT, since they are changes rather than replacements. Let's look at each service in more detail.

Reading categories list

An XML list of categories is exposed by GETting <http://ajaxify.com/run/shop/categories.phtml>. Here, categories is the resource and the HTTP method is GET since we're reading the resource. To avoid listing all category information here, there's a link to each specific category resource:

```
<categories>
  <category xlink="http://ajaxify.com/run/shop/rest/category.phtml?name=Books">
Books</category>
  <category xlink="http://ajaxify.com/run/shop/rest/category.phtml?name=Songs">
Songs</category>
  <category xlink="http://ajaxify.com/run/shop/rest/category.phtml?name=Movies">
Movies</category>
</categories>
```

Reading an individual category

To drill down to an individual category, say "Movies," you GET <http://ajaxify.com/run/shop/category.phtml?name=Movies>, which provides the name and items. Since an item is defined solely by its name, and we can't perform operations on items themselves, there's no need to give it a URL.

```

<category>
  <name>Movies</name>
  <item>Contact</item>
  <item>Gatica</item>
  <item>Solaris</item>
</category>

```

As the system scales up, the list of all items gets excessive for someone who just wants to know the category name. So the next service provides just the items. If we want, we could then remove the list of items in the category name.

Reading cart contents

Being RESTful, we can't just access the shopping cart out using the current session—each operation must specify the cart's owner. The session is used here, but only for authentication.

To read the shopping cart of user 5000, we GET <http://ajaxify.com/run/shop/rest/cart.phtml?userId=5000>:

```

<cart userId="5000">
  <item>
    <name>Hackers and Painters</name>
    <amount>2</amount>
  </item>
  <item>
    <name>Accidental Empires</name>
    <amount>4</amount>
  </item>
</cart>

```

To test the authentication, visit the web app <http://ajaxify.com/run/shop/rest> and add a few items. Cut and paste your assigned user ID over the “5000” in the above URL. You'll be able to see your cart. Then try a different user ID, say 6000, and you'll be greeted with the following message along with a 401 (Forbidden) header:

```
You don't have access to Cart for user 6000.
```

Changing cart contents

Being RESTful, the cart URL remains the same whether we're manipulating or reading it. So when we make changes to user 5000's cart, we use the same URL as for reading it, <http://ajaxify.com/run/shop/rest/cart.phtml?userId=5000> (which would instead end with the cleaner `/cart/5000` if we could use URL-rewriting).

To update the cart, we simply PUT a new cart specification to the cart's URL. If the user's just cleared the cart, the following will be uploaded:

```

<cart>
</cart>

```

If the user's added an item, the browser still does the same thing: just upload the whole cart; e.g.:


```

<cart userId="65590">
  <item>
    <name>Hackers and Painters</name>
    <amount>2</amount>
  </item>
  <item>
    <name>Accidental Empires</name>
    <amount>5</amount>
  </item>
</cart>

```

Note that working with PUT is quite similar to working with POST. As explained in *XMLHttpRequest Call* (Chapter 6), XMLHttpRequest has a `requestType` parameter. In the example here, the details are in any event abstracted by the `AjaxCaller` wrapper library:

```

ajaxCaller.putBody("cart.phtml?userId="+userId, null, onCartResponse, true,
  null, "text/xml", cartXML);

```

How to mail the cart contents is more subjective, as discussed in the “Arbitrary Actions” section in the Solution. The approach here is to post an email address to the cart URL, e.g.:

```

<email-address>
  ajaxmail@example.com
</email-address>

```

SUPPORT FOR PUT AND DELETE

Supporting PUT and DELETE in the server depends on the programming environment. In Java, for example, it's trivial because servlets contain the methods `doPut()` and `doDelete()` alongside `doPost()` and `doGet()` methods. PHP, on the other hand, gives special treatment to GET and POST. Not only are variables exposed with the `$_GET` and `$_POST` arrays, but the POST body is accessible with `$HTTP_RAW_POST_DATA`. To read the body of a PUT call, the service here has to read from the input stream, `php://input` (<http://php.net/wrappers.php>):

```

function readBody() {
    $body="";
    $putdata = fopen("php://input", "r");
    while ($block = fread($putdata, 1024)) {
        $body = $body.$block;
    }
    fclose($putdata);
    return $body;
}

```

It's partly inconveniences like this that drive some developers to compromise and use POST as a substitute for PUT and DELETE.

Alternatives

RPC Service

RPC Service (see later in this chapter) is an alternative to RESTful Service, as highlighted earlier in the “Solution.”

Related Patterns

XML Message

XML Messages (see later in this chapter) are often used as the format of a REST service’s response, and sometimes its input as well. XML fits well here because it’s standards-based, broadly supported and understood, self-documenting, and capable of self-verifying when used with DTDs and XML Schemas. XML Messages tend to appear as the body of PUT and POST calls as well as in responses.

XML Data Island

A RESTful transaction often involves passing a resource back and forth, with each side augmenting its state. For instance, a server can deliver an initially empty shopping cart, the client can add an item to it, the server can set the total price, and so on. Instead of transforming to and from a custom data structure, it sometimes makes sense for the browser to retain the incoming *XML Messages* in an *XML Data Island* (Chapter 11) and transform it according to the interaction.

Unique URLs

Unique URLs (Chapter 17) also relates to URL design, though they differ in scope. Unique URLs relates to the URLs of the Ajax App itself, while RESTful Service involves the URLs of web services it uses. It’s sometimes said that “Ajax breaks REST” because many Ajax Apps have a single URL regardless of state. Ironically, though, Ajax actually facilitates REST by encouraging a clean API to service the browser application. As for the browser application, Unique URLs do help make an Ajax App a bit more RESTful, though that’s somewhat beside the point since it’s the web services that a client would want to use.

Metaphor

REST is often likened to Unix—with its consistent usage of pipes and filters—and to SQL—with its standard persistence and querying commands—but can also be compared to Windows-based GUI platforms. There are standard actions like “Cut,” “Paste,” and “Popup Context Menu.” Each object (e.g., documents, icons, and folders) accepts such actions and interprets them in a manner consistent with the action’s meaning.

Want To Know More?

- Roy Fielding’s Dissertation that introduced the REST Concept (<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>)
- REST Wiki (<http://rest.blueoxen.net/cgi-bin/wiki.pl>)

- Jeff Bone's breakdown of REST vs. RPC (<http://www.xent.com/pipermail/fork/2001-August/002801.html>)
- Roger Costello's "Building Web Services the REST Way" (<http://www.xfront.com/REST-Web-Services.html>)
- PUT versus POST discussion (<http://www.megginson.com/blogs/quoderat/archives/2005/04/03/post-in-rest-create-update-or-action/>)

Acknowledgments

Thanks to Tony Hill from the Thomson Corporation for reviewing this pattern and providing some great insights into REST.

RPC Service



Delegate, Facade, Procedure, Proxy, Remote, Remoting, RPC, SOAP, WSDL

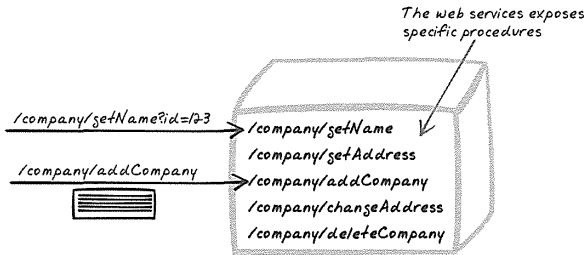


FIGURE 9-2. RPC Service

Developer Story

Dave's finished coding the backend services for some blogging software. There's an `ArticleRepository` object, for example, with functions to create, remove, update, and delete articles. Using a new product, he's able to effortlessly expose `ArticleRepository` as a web service API, with a different URL for each operation. Now he can code up the browser script for his own application to talk to the API, and there's a ready-made API available to third parties too.

Problem

How do you expose services on the Web?

Forces

Refer to the section "Forces" in *RESTful Service*.

Solution

Expose web services as Remote Procedural Calls (RPCs). Like REST, RPC is⁶ a broad term. It's somewhat ambiguous too, meaning different things to different people. This pattern uses a fairly general definition—a Remote Procedural Call (RPC) is a form of communication where the client invokes a remote procedure on the server. The definition implies:

- RPCs are generally characterized as actions. The URL is usually verb-like; e.g., `/game/createGame?gameId=995` or `/game/updater?command=createGame@gameId=995`.
- Typically, though not necessarily, the remoting is designed to be relatively transparent. The server's backend is developed, and a remoting framework acts as a bridge between client and server, allowing the client to directly invoke backend operations. Often, the remoting framework is distributed across both tiers, wrapping and unwrapping messages at each end.

Here are some concrete forms of RPC:

Procedure-style API

In the simplest case, it's just a matter of the URL style: the server exposes URLs that look like verbs and invite action, though they don't necessarily have any direct relationship to the backend technology.

XML-RPC

XML-RPC is a simple protocol for RPCs, using tags like `<methodCall>`, `<methodName>`, and `<params>`. An XML document is uploaded to make the call, and another document is returned with the response. There are many frameworks available to automatically expose backend services as XML-RPC and to invoke such services, so the remoting is fairly transparent from the developer's perspective.

SOAP

SOAP is based on exposing selected backend operations as HTTP-based services that can be accessed by certain clients. It's similar to XML-RPC, but offers a whole slew of additional functionality, such as custom data types and asynchronous messaging. The protocol is intended to automatic translation of SOAP calls to and from calls in the native languages being used. For instance, Enterprise Java Beans can automatically be exposed as web services using fancy code generators or mediating proxies. Unfortunately, the complexity of SOAP comes at a cost, and using it can sometimes feel like driving in a nail with a two-ton sledgehammer; some jaded developers have wondered whether SOAP is more about selling consulting services and toolkits than actually making life any easier.

Ajax Stub frameworks

These frameworks are an all-in-one package. XML-RPC and SOAP might automatically generate a web service for you, but you still have to invoke it yourself from an Ajax App. In contrast, *Ajax Stub* (see later in this chapter) frameworks such as SAJAX (<http://modernmethod.com/sajax>) create JavaScript wrappers for server-side operations

as well as exposing them in the first place. The JavaScript code doesn't have to issue *XMLHttpRequest Calls* directly or even with an *XMLHttpRequest* wrapper. It instead calls a generic remoting function, capable of invoking operations on the backend. See the Refactoring Illustration in the section *Ajax Stub*, later in this chapter, for an example. The downside is that *Ajax Stubs* tend not to be fussed about exposing a clean API, because it's assumed the programmer will never need to look at how the server's invoked. Consequently, if a third party needs to use it, the API will quite possibly be uglier than an auto-generated XML-RPC or SOAP API.

Can RPC be RESTful? Some argue it can. *RESTful Service* is one of the largest patterns in this language, precisely because REST is a broad idea consisting of many principles and conventions. Given that RPC URLs represent actions rather than resources, it's a little over the top to say that RPC can be made truly RESTful. However, many of the REST principles do make sense in an RPC context, and it's advisable to follow them. Remember that many entities on the Internet, such as caches, are based on certain RESTful assumptions, and you'll be affected by those no matter how you design your interface. A few guidelines:

- Read-only queries should be made as GET requests, meaning that you shouldn't pass a query in the body.
- Anything that changes server state should be made with POST, including the command and arguments in the body. The command will already indicate the type of action that's occurring, so there's probably no need to use other HTTP methods such as PUT or DELETE.
- Stateful conversation is best avoided. Calls should pass in all the information they need explicitly. Session and cookies should not determine how the server responds to queries; if used at all, they should only be used for authentication.

Real-World Examples

Kiko

Kiko (<http://kiko.com>) is an online calendar application with a slew of Ajax features. The browser communicates to an interface using RPC-style URLs. For example, the following call retrieves my contacts:

```
http://www.kiko.com/kiko/kikoservlet?function=SelectContactsByUser , false ,  
undefined , undefined)
```

A change is made in the same way. Here's what the browser called when I added an appointment:

```
http://www.kiko.com/kiko/kikoservlet?function=CreateAppt&starttime=2005-9-12 18:0:  
00&endtime=2005-9-12 18:30:00 , false , undefined , undefined)
```

Kiko invokes these URLs using POSTs, where the body is just an authentication key. Note that the service could be more REST-like, while retaining the RPC-style URLs, if Kiko wants to open the API to the public and ensure scalability. To make it more REST-like,

queries like `SelectContactsByUser` would be issued as GETs, with cookies used for authentication in place of the authentication key in the body. This would not only improve consistency across APIs, but also have the direct practical benefit of supporting caching, since GET responses can easily be cached. Also, POSTs could include all arguments inside the body, while retaining RPC-style URLs like `/?function=CreateAppt`.

Flickr API

The Flickr API (<http://www.flickr.com/services/api/>), available to external developers, is a good example of an API with RPC-style URLs that also respects the basic REST conventions of GET strictly for reads and POST strictly for writes. Most calls include an API key, which the developer must apply for. A query for photo details such as owner and description is a GET call to a URL like this:

```
http://www.flickr.com/services/rest/?method=flickr.photos.getInfo&api_key=ap1000&photo_id=2000
```

You can also effect changes with the API. For example, you can tag an element by POSTing to a URL like this:

```
http://www.flickr.com/services/rest/?method=flickr.photos.addTags
```

containing a body like this:

```
api_key=ap1000&photo_id=2000&tags=Screenshot
```

Code Example: AjaxPatterns RPC Shop Demo

The Basic Ajax Shop Demo (<http://ajaxify.com/run/shop>) provides a web service to expose and manipulate store items and user shopping carts. In this demo, the cart and product services are refactored to provide a cohesive RPC service. You can contrast this to the Refactoring Illustration in *RESTful Service*.

A single service acts as a Facade (Gamma et al., 1995) to all server-side functionality. It accepts command-like URLs and routes them to the appropriate function. Read-only queries are of the following form, with the argument being optional:

```
services.phtml?command=queryType&someArg=someValue
```

State-affecting commands instead require all arguments, including the command, to be posted to a simple URL:

```
services.phtml
```

Because the URL is generic, the command must qualify what object it applies to; e.g., `addToCart` as opposed to `add`.

Reading the Categories list

To read the Categories, you GET `http://ajaxify.com/shop/rpc/services.phtml?command=get-Categories`.

Reading an individual category

To drill down to an individual category, say “Movies,” you GET `http://ajaxify.com/shop/rpc/services.phtml?categoryName=Movies`.

Reading cart contents

To read cart contents, you GET `http://ajaxify.com/shop/rpc/services.phtml?command=getCart`. Note that cart access is based on the session, as with the Basic Shop Demo (and unlike the *RESTful Service* demo). It doesn’t have to be like this, but doing so allowed for a minimal change.

Changing cart contents

To clear the cart, we POST a body of `command=clearCart` to `services.phtml`. To add an item, we POST the command and item in the body, such as `command=addToCart&item=Accidental Empires` to `services.phtml`.

Alternatives

RESTful Service

RESTful Service (see earlier in this chapter) is an alternative to RPC Service, but many tenets of REST can, and should, still be followed in RPC. See the *RESTful Service “Solution”* for a comparison.

Related Patterns

Ajax Stub

Ajax Stub (see later in this chapter) automates the production of RPC Services.

Plain-Text Message, XML Message

RPC is often used to design an application-independent web service API, or at least one that doesn’t know anything about the user interface. That being the case, responses tend to be of a raw, semantic nature such as a *Plain-Text Message* (see later in this chapter) or an *XML Message* (see later).

Metaphor

Task-oriented user interfaces, such as simple menu systems and the new MS Office Task Ribbon idea, are similar to RPC. They explicitly offer the typical tasks you wish to achieve.

Ajax Stub



Delegate, Facade, Procedure, Proxy, Remote, Remoting, RPC

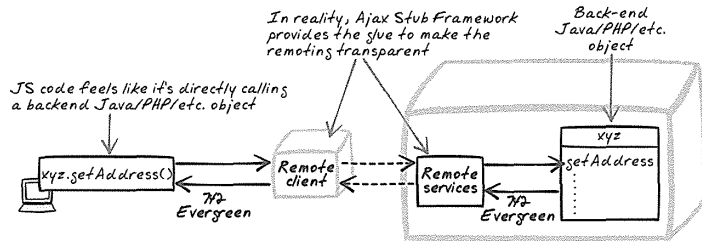


FIGURE 9-3. Ajax Stub

Developer Story

Devi begins working on the browser script for a trading application and soon decides she needs a way to execute a trade. There's already a method to do that on the backend TradeManager object, so after 15 seconds reconfiguring the Ajax Stub, the browser script is able to invoke the method.

Problem

How do you implement an *RPC Service* (see earlier in this chapter)?

Forces

- Some services have to reside on the server, for reasons of security, complexity, and performance. Thus, you need to expose web services to be accessed from the browser script.
- A web service should encapsulate only web-related logic and delegate any business and application logic to the backend.
- This creates a redundancy: the web service and the backend logic are tied together. When one changes, the other must change, leading to more work and the risk that you'll forget to change one or the other.
- The browser needs to access web services via XMLHttpRequest over HTTP, which is fundamentally different to the way both JavaScript and server-side scripts work. It's tedious to wrap and unwrap messages into a form suitable for transfer.

Solution

Use an *Ajax Stub* framework that allows browser scripts to directly invoke server-side operations, without having to worry about the details of **XMLHttpRequest** and **HTTP** transfer. The aim is to let the JavaScript call remote operations as if they were regular JavaScript functions, facilitating a more natural program style.

There are several frameworks to support stubbing of this nature. Typically, you declare which backend operations should be exposed, and the framework takes care of the remoting. It will provide a JavaScript stub object that will delegate the call over to the server side. For example, you might have a Java method like this:

```
public class Book {
    public String getEdition(int year) {...}
    ...
}
```

After configuring an Ajax Stub framework for handling remoting, it's light work to make call the class from within the browser:

```
editionLabel.innerHTML = book.getEdition();
```

Typical facilities include:

- Ability to convert JavaScript arguments and return values into server-side language constructs
- Sensible handling of server-side errors and exceptions
- Support for general XMLHttpRequest issues, such as timeouts

A stubbing framework usually consists of two components:

- A JavaScript component marshalls the call into an XMLHttpRequest, handles the call, and later unmarshalls the response.
- A server-side configuration component, with some configuration, that accepts the call, delegates to the backend operation, and marshalls the return value into a suitable response.

Decisions

How will you secure the server-side function?

With great power comes great responsibility. An Ajax Stub makes it easy, bordering on trivial, to directly expose business operations. That means it's also easy to let curious users do things they shouldn't. Since many Ajax Apps push the whole user interface out to the browser, the web tier is left publishing business methods. Harry Fuecks, author of the Ajax Stub toolkit JPspan (<http://jpspan.sourceforge.net/>), gave this cautionary example on the Ajax Blog (<http://ajaxblog.com/archives/2005/05/25/a-grumpier-ajaxian>). A naïve export of a business operation could end up with a call like this:

```
var ccNum = AJAX.getCreditCardNumber(userId);
```

Anyone can edit the JavaScript and pass any argument to the credit card operation. Another stubbing framework, DWR (<http://getahead.ltd.uk/dwr/getstarted>), warns: "There is a danger that you could cause all sorts of security problems using this code. You need to think about security earlier rather than later."

Other approaches, such as *RESTful Service*, are also vulnerable to the same threat, but they tend to encourage more analysis, because you have to explicitly design the service interface. With Ajax Stub, you're effectively ticking a few boxes to say which backend operations can be executed.

A few guidelines:

- Even though you *can* publish an interface very quickly, that doesn't mean you should. Do take the time to consider the implications of making an operation available to the public (or whatever user base will be using your Ajax App), who will be able to call it with whatever arguments they please.
- The framework's module is exposed as a web service—your environment might be able to control access to particular services. For example, the DWR Framework for Java stubbing (<http://getahead.ltd.uk/dwr/security>) notes: "DWR allows you to grant access using two J2EE based mechanisms. Firstly you can define access to dwr based on J2EE roles. Secondly within DWR you can define access to methods on a role basis."
- Depending on the environment, the backend operation may need to use some form of authentication. In some cases, the operation will have direct access to request information such as the URL and cookies, which will facilitate authentication. The downside is that you've begun to incorporate web-related concepts into the business logic, which is what Ajax Stubs seek to avoid.
- Be wary of the framework as well. One Ajax Stub framework, CPAINT (<http://cpaint.sourceforge.net/>), caused some security alerts due to the possibility of malicious code being executed on the server (<http://secunia.com/advisories/16454/>). The team was quick to respond, and the Ajax Stub pattern is not inherently flawed, as some people had assumed. However, it's important to understand just what exactly the Ajax Stub framework is doing.

Real-World Examples

These examples cover available frameworks for creating Ajax Stubs.

SAJAX framework

SAJAX (<http://www.modernmethod.com/sajax/>) is an Ajax Stub framework that actually supports multiple backend languages. The browser side is the same for all, but there are different server-side proxy components depending on which language is used. Among the backend languages are ASP, ColdFusion, Perl, PHP, Python, and Ruby.

DWR framework

DWR (<http://getahead.ltd.uk/dwr/overview/dwr>) is an open source framework for Ajax Stubs to Java classes. On the server side, you run a DWR servlet and configure it with a separate XML file. In the browser, just include a JavaScript file. Java and JavaScript may not have much in common, but calls do at least look the same. A DWR JavaScript-to-Java call looks just like a Java-to-Java call.

CL-AJAX framework

Richard Newman's open source CL-AJAX (<http://www.cliki.net/cl-ajax>) supports stubbing of Common Lisp functions.

Code Refactoring: AjaxPatterns SAJAX Sum

In this refactoring illustration, the Basic Sum Demo (<http://ajaxify.com/run/sum>) is refactored to use an Ajax Stub (<http://ajaxify.com/run/sum/stub>), based on SAJAX (<http://www.modernmethod.com/sajax/>). To begin with, we do away with the old Sum service (`sum.phtml`) and create a standalone, backend, calculator module. Now that we're using an RPC approach, we'll call the operation `add` instead of `sum`, to make it distinctly a verb. So `calculator.phtml` looks like this:

```
<?
function add($figure1, $figure2, $figure3) {
    echo $figure1 + $figure2 + $figure3;
}
?>
```

With the stub, we can allow the browser-side script to directly invoke that function. The JavaScript call looks like this:

```
function submitSum() {
    x_add($("#figure1").value,$("#figure2").value,$("#figure3").value,onSumResponse);
}
```

Here's how you set up the SAJAX glue between the JavaScript `x_add` and the PHP `add`. In PHP, we tell SAJAX which functions are exported:

```
<? require_once("Sajax.php"); ?>
...
sajax_init();
sajax_export("add");
sajax_handle_client_request();
```

And in outputting the script, some PHP code generates the required JavaScript—in this case, the `x_add()` function, which will forward on to SAJAX on the server side and eventually interpret its response:

```
<script>
<?
    sajax_show_javascript();
?>
</script>
```

Alternatives

XMLHttpRequest Call

Ajax Stubs are built on top of *XMLHttpRequest Calls* (Chapter 6) and represent a different approach to the usual Ajax remoting. XMLHttpRequest Calls leave the user to deal directly with HTTP concepts and packing and unpacking of messages, tasks that Ajax Stubs take

care of. The benefit of direct XMLHttpRequest Calls is that the server-side service interface is independent of the underlying implementation.

XML-RPC and SOAP

XML-RPC and SOAP are more generic (non-Ajax-specific) ways to develop *RPC Services* (see earlier in this chapter). Since Ajax Stubs tend to produce Ajax-specific services, XML-RPC or SOAP might be more appropriate if you're opening up to external clients.

Related Patterns

RPC Service

An Ajax Stub is one way to implement an *RPC Service*.

JSON Message

JSON Messages (see later in this chapter) are being used by several Ajax Stub frameworks because Ajax Stubs need to share data between browser and server; i.e., inbound arguments and outbound return values. These objects should be standard JavaScript objects in the browser and standard Java or PHP or whatever-your-language-choice objects on the server. In between, it's most convenient to transfer them as strings. JSON defines a standard way to convert objects in many languages to and from a string representation.

An even more ambitious standard is JSON-RPC (<http://json-rpc.org/>), a protocol for remototing that's touted as a lightweight alternative to XML-RPC. There's an Ajax Stub framework based on JSON-RPC, the aptly named JSON-RPC-Java (<http://oss.metaparadigm.com/jsonrpc/>).

HTML Message



AAAH, Direct, Display, HTML, InnerHTML, Message, Precise, ServerSide, Visual

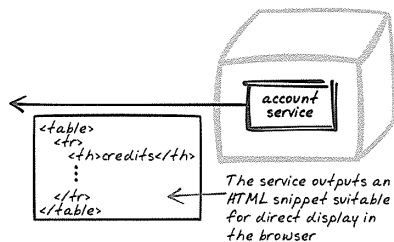


FIGURE 9-4. HTML Message

Developer Story

Dave's identified the need for a credit history service, providing a list of transactions. JavaScript resources are limited, so the entire HTML is created server side, and the browser application has only to morph a DOM element with the entire HTML response.

Problem

What format should be used for server responses?

Forces

- The browser display needs to be dynamic in Ajax Apps.
- The display changes by altering the DOM, and HTML is often used to specify the new value.
- The nature of the change is often complex and needs to be determined server side.
- Not all developers know JavaScript, and many feel its usage is best minimized.

Solution

Have the server generate HTML snippets to be displayed in the browser. In this approach to browser-server dialogue, the server-side service outputs some HTML, which is picked up by the XMLHttpRequest callback function and an element then morphed by setting its `innerHTML` property to the response HTML. In general, the server side is application-specific because the HTML response is closely tied to the application's display style.

The XMLHttpRequest allows callers to retrieve responses as either XML or plain-text. With HTML, it's usually easiest to retrieve it as a plain-text string. If the HTML is a pure XHTML document, XML is also an option, but usually that's not the case because there's no need for a header section. The entire response can be as simple as "`<strong>You Win!</strong>`". That's not an XHTML document because there's no `<xml>` tag, nor are there HTML header and body sections.

HTML Messages should be used with caution because they couple server-side services with browser display. That means it's difficult to develop the tiers in parallel. In maintenance, if you change the display by altering its initial HTML or any JavaScript manipulation, you often need to change the HTML-generating service too. There's also a risk on the server side that you might be coupling business logic with HTML generation. So when might an HTML Message be appropriate?

Performance

Since browser-side parsing and rendering can be expensive, HTML Messages can make an application more responsive, especially if the HTML is cached on the server or the browser.

Server-Side Code Generation

One case where HTML Messages make some sense is with *Server-Side Code Generation* (Chapter 12), where the server builds all the browser-side code for you. There are

strengths and weaknesses of such frameworks, and if you decide to use one, then browser-server coupling is not really an issue because all maintained code is server side anyway.

Legacy Code

Most legacy applications will use the conventional approach of publishing pages from the server. They therefore already have all the HTML generation present in the server-side environment, so if you're Ajaxifying a legacy application, a quick migration path is to retain the server-side HTML generation where possible.

Complex HTML

In the rare case that your HTML or JavaScript is particularly complex, you might prefer to generate it all on the server side, where development and debugging is sometimes easier.

Graceful Degradation

When there's a chance users won't have a recent browsers, HTML Messages allow you to encapsulate more logic in the server. Of course, if JavaScript is not enabled at all, the "messages" will have to be full pages.

Server-Centric Attitude

If you're concerned about keeping as much logic as possible in the one language and environment, or you'd prefer not to work much with JavaScript, HTML Messages are more appropriate.

Typically, HTML Messages rely on a block-level element existing in the DOM, often a div or form. The HTML Message will contain a specification for the entire contents of the element. Some of the XMLHttpRequest wrapper libraries, such as JAH/AHAH (<http://microformats.org/wiki/rest/ahah>) and Sack (<http://twilightuniverse.com/2005/05/sack-of-ajax/>), support HTML Messages by letting the caller directly state a DOM element as the callback destination instead of the usual callback function. When the response returns, the DOM element is automatically morphed with the service's response.

A variant of this pattern is Behavior Message; see *On-Demand JavaScript* (Chapter 6). Also, note that HTML Messages almost flow from server to browser; you'd rarely need to upload an HTML Message back to the server.

Decisions

At what level of granularity will the HTML apply?

In the extreme, the service could generate HTML for the entire document, making it similar to a conventional page refresh. In practice, you'll usually want to limit the HTML's level of granularity to well-defined, distinct page elements. For example:

- An account status
- A form
- A menu

How much style will be contained in the message?

You'll need to decide how much style is contained in the message. An HTML Message already ties the server to the browser somewhat, but style directives will tie it further to the server. In general, it's good practice to just include class names and element IDs so that the browser application can influence the style, usually with a standard CSS stylesheet. If you do this, you'll need to ensure the names are unique, and you'll need to decide on exactly which elements need to be given ID and class declarations.

Real-World Examples

Digg Spy

Digg Spy (<http://digg.com/spy>) shows new stories and events such as user moderation as they happen (see Figure 5-5). The *XMLHttpRequest Call* (Chapter 6) accesses a static URL (<http://digg.com/last10>) containing the last 10 submissions and integrates them into the page. The HTML looks like this:

```
<div class="news-body" id="main64115">
  <h3 id="title">
    <a href="http://digg.com/movies/How_LEGOs_Are_Made_"> How LEGOs Are Made! </a>
  </h3>
  <p class="news-submitted">
    <a href="/users/Akshun"> </a>
    submitted by <a href="/users/Akshun"> Akshun </a> 14 hours 22 minutes ago
    (<a href="http://www.popandco.com/archive/moab/" class="simple tight" title=
    "How LEGOs Are Made!"> ... </a>) </p>
  <p> </p>
</div>
<div class="news-body" id="main64550">
  ...
```

Rapha

Rapha (<http://www.rapha.cc>) is an e-commerce web site for cycling gear (Figure 9-5). The shopping cart is Ajaxian, requiring no page refresh to update. Each time an item is purchased, the HTML for the shopping cart is retrieved. For example:

```
<div id="basket">
  <h3>Your basket:</h3>
  <ul>
    <li>Small Softshell Jacket x 2</li>
  </ul>
  <p><a href="/basket/">Edit selection</a> | <a href=
  "/checkout/">Go to checkout</a></p>
</div>
```

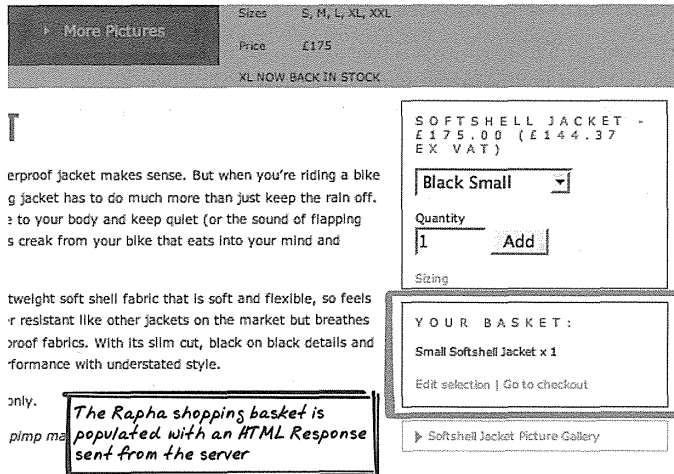


FIGURE 9-5. Rapha

Amazon Zuggest

Francis Shanahan's Amazon Zuggest (<http://www.francisschanahan.com/zuggest.aspx>) provides a *Live Search*: results are shown as you type. The server responds by providing the entire HTML for each result—for example:

```
<table>
  <tr><td valign=top width='20%'><b>Vempire Or Dark Faerytales in Phallustei</b><br>
  <a href='http://www.amazon.com/exec/obidos/redirect?tag=francshanacom-20%26link_
code=sp1%26camp=2025%26creative=165953%26path=http://www.amazon.com/gp/redirect.
html%253fASIN=B0000525ZY%2526tag=francshanacom-
20%2526lcode=sp1%2526cID=2025%2526ccmID=165953%2526location=/o/ASIN/
B0000525ZY%25253FSubscriptionId=16KBB0XN5XP4WSNNVKG2'
target=_blank>Click to View</a><br>
  [Music]<br>List Price<span class='lp'>$23.99</span><br>
  <span class='lnp'>1 NEW from $14.99[$13.50 used]</span>
  ...
</td></tr><table>
```

TalkDigger

TalkDigger (<http://talkdigger.com>) is a webfeed meta-search. Enter a query, and it fires off parallel queries to different search engines (an example of *Multi-Stage Download*). Each result is returned as an HTML table to be added to the page.

Code Example: Digg Spy

Digg Spy (<http://digg.com/spy>) responds with the full HTML to be displayed as well as some meta-content at the end. The results are fetched periodically—`filldigs()` is triggered by

the recurring `lastdigs()`. `filldigs()` executes an *XMLHttpRequest Call*, extracts the HTML from the full response, and morphs the results panel using its `innerHTML` property:

```
function startlastdigs() {
    window.setInterval("lastdigs()",15000);
}
function filldigs() {
    ...
    s.open("GET",url2,true);
    s.onreadystatechange=function() {
        if (s.readyState == 4) {
            ...
            b = responsestring2.split(split);
            ...
            document.getElementById('diggsby').innerHTML = b[0];
            ...
        }
    }
    ...
}
```

Alternatives

Plain-Text Message, XML Message, JSON Message

Plain-Text Message, *XML Message*, and *JSON Message* (see more on these later in this chapter) all involve sending raw data as opposed to concrete display detail. When the browser receives such responses, it can render it or use it in some other way, such as retaining some information. In contrast, HTML Messages are likely to be display directly and are not retained. A further difference is that HTML Messages tend to be responses only, whereas these other formats are often used for uploading data as well.

Related Patterns

On-Demand JavaScript

Where HTML Message provides snippets of HTML to be fused on to the page, *On-Demand JavaScript* (Chapter 6) can provide snippets of JavaScript to be executed immediately. The idea is introduced in that pattern as “Behavior Message.”

Metaphor

Sending an HTML Message is like sending a photograph—there’s a lot of visual detail, but any metadata, such as location and subjects, can only be inferred.

Plain-Text Message



Acknowledgement, Custom, PlainText, Semantic, String

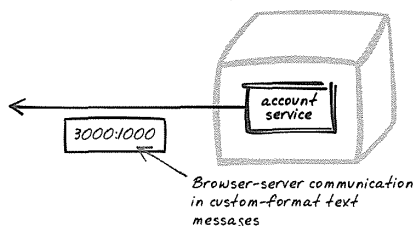


FIGURE 9-6. Plain-Text Message

Developer Story

Devi's working on an auto-completion form field. Every few seconds, the server needs to respond with a list of suggestions. Devi codes the server side to output the suggestions as a comma-separated list, as it's convenient to output on the server and easy to parse in the browser.

Problem

How can you transfer data between server and browser?

Forces

- Ajax Apps require messages to be transmitted back and forth.
- Both browser and the server must be able to access the message. That usually means the format must be easily accessed in JavaScript as well as in whichever server-side language is used.

Solution

Pass simple messages between server and browser in plain-text format. This is a broad pattern, because "plain-text" is an ambiguous term. The emphasis here is on keeping things simple—XML or JSON are often more appropriate for complex data structures, but can also add unnecessary complexity.

Despite the name, XMLHttpRequest is perfectly capable of working with plain-text responses. As discussed in *XMLHttpRequest Call* (Chapter 6) and *XML Message* (later), the browser script can decide how to interpret the response in XMLHttpRequest by switching between the `responseText` and `responseXML` properties.

Plain-text formats include:

Response Code

Sometimes, all that's required is a simple response code, such as a numeric ID or a text message such as "OK" or "Failed." Note that, in this case, an appropriate HTTP Response Code codes may be more standards-based and in line with *RESTful Service* conventions.

A simple text string

For example, a user's name.

A list

For example, a comma-separated list of search results.

A custom data format

For example, a list of strings, where each string is a comma-separated list of object attributes.

While browser support for XML is reasonably good, especially with third-party libraries, it's also quite easy to manipulate arbitrary text formats. JavaScript provides a standard string manipulation library, including the particularly helpful regular expression and grouping facilities.

The case for plain-text is particularly compelling in the following situations:

Simpler messages

The simpler the message, the more compelling the argument for plain-text messages. Once the string starts to get more complex, there's a better argument for relying on XML support from the browser and third-party libraries.

External clients

XML is often a better choice when it's not just the web app that's using the service. That's partly for reasons of convention—XML is simply a standard transfer format on the Internet. But there's also a more technical reason: using DTDs or XML Schemas, you can specify the format in an unambiguous manner that can be enforced whenever a document is encountered.

Skill base

XML may be a standard, but that's useful only if people know the standard. The better developers are with XML technologies in both the browser and the server, the more compelling the argument for XML.

Real-World Examples

Lace Chat

Brett Stimmerman's Lace Chat (<http://www.socket7.net/lace/>) is an Ajax chat app (Figure 9-7). When the page is initially shown, all user messages are downloaded as a single, long string. Within the string, each user message is captured as a set of attributes, separated by pipe characters:

```
1135646360:749|||date-1135646204||*Monday, 26 December
2005|||hour-17||*17:00|||1135646204||Posted about 3 minutes ago at
17:16||Aj Two||Yeah you can *SO* chat in real time|||1135646212||Posted
about 2 minutes, 52 seconds ago at 17:16||*Lace||AJ One
has joined the conversation|||1135646212||
```

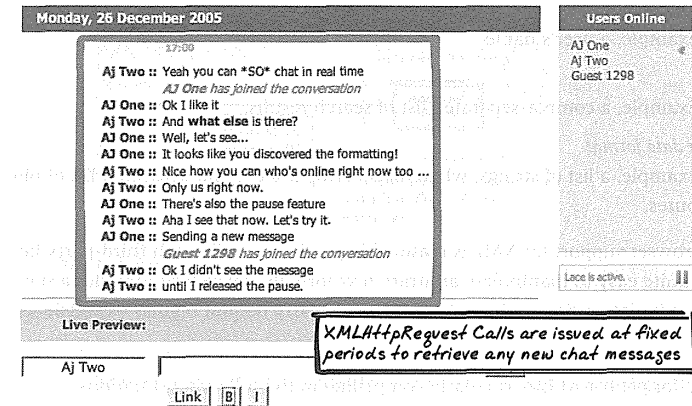


FIGURE 9-7. Lace Chat

Magnetic Poetry

In Magnetic Poetry (<http://www.broken-notebook.com/magnetic/>), you drag tiles around a workspace. When the browser receives an updated position, the message looks like a structured command statement:

```
Update magnetic_poetry set top_value=291, left_value=119 WHERE id=81
```

HousingMaps

HousingMaps (<http://housingmaps.com>) is a mashup between Google Maps and Craigslist (<http://craigslist.com>), a community-oriented classified advertising web site. The HousingMaps portion is downloaded with an *XMLHttpRequest Call* that retrieves the 15 or so results matching a query. Each result represents a single home and is broken into a sequence of attributes, one line at a time, and prefixed with an identifier:

```
PA:34.0917
PN:-118.28
IC:p
DS:One Bath With Small Loft Duplex Near Sunset Junction
AD:Hyperion Ave &&& Sunset Blvd</line><line>Los Angeles</line><line>CA
...
I4:a.im.craigslist.org/rR/iv/X1dT18TzV07cfjuQzouhKQ5EETuK.jpg
LC:1071999
PA:34.0763
PN:-118.294
IC:p
DS:1930&&#39;s Restored Spanish Beauty- 1st Month Free
AD:Beverly &&& Beverly</line><line>Los Angeles</line><line>CA
...
```

```
I4:a.im.craigslist.org/n0/Xm/SS3hSoIWL0sc8w0jailGQK5Hfbzl.jpg
LC:1072004
...
```

Code Example: Lace Chat

The response in Lace Chat is a list of attributes, separated by pipe characters. Each complete record is separated by ||||, and each attribute within a record is separated by a smaller string, ||. The handling code first splits the string into an array of records, then splits out the attributes within each record:

```
// Records are separated by four pipes ||||
var results = this.httpGetObj.responseText.split('||||');
...
for (var i = 1; i < results.length; i++) {
    var first;
    // Fields are separated by two pipes ||
    var fields = results[i].split('||');
    ...
    var timeStr = fields[1];
    var textStr = fields[3];
    ...
    p.setAttribute('id', 'msg_' + fields[0]);
    ...
}
```

Alternatives

XML Message

As discussed in the “Solution,” an *XML Message* may be a better alternative for more complex data.

XML Message



Document, Format, Hierarchical, PlainOldXML, POX, Semantic, Structured, XML

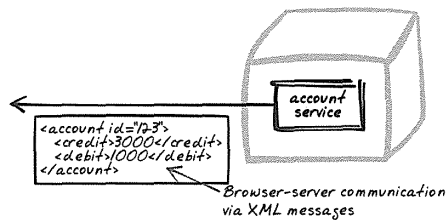


FIGURE 9-8. XML Message

Developer Story

Dave is creating an Ajax search engine. The results are offered as a *RESTful Service*, so that external developers can create their own interface. The results are presented as XML, and Dave's own search interface pulls it down with an XMLHttpRequest call, then transforms the result to HTML.

Problem

How can you transfer data between server and browser?

Forces

- Ajax Apps require messages to be transmitted back and forth.
- Both browser and the server must be able to access the message. That usually means the format must be easily accessed in JavaScript as well as whichever server-side language is used.

Solution

Pass messages between server and browser in Extensible Markup Language (XML) format. It's feasible to pass XML in both directions. On the server side, it's a common way to expose structured data as text, and browser-side applications can download XML with an XMLHttpRequest Call and parse it in different ways. Where the browser needs to upload some complex data, it's fairly easy to serialize the data as XML and upload it in the body of an XMLHttpRequest Call.

XML is the definitive standard for data transfer across the industry, so just about any server-side development language has comprehensive support. The great advantage of XML over custom formats is the massive base of supporting frameworks, libraries, editors, and general tools. How you deal with XML on the server is language-specific, and a few choices are outlined in "Decisions," next. However you do it, make sure the server presents XML content with the appropriate header to ensure its handled correctly by XMLHttpRequest:

```
Content-type: text/xml
```

As the name implies, XMLHttpRequest was developed with XML transfer in mind. To retrieve an object with an XMLHttpRequest, first point it to a URL known to serve XML:

```
xReq.open("GET", "http://ajaxify.com/run/portal/drilldown/drilldown.phtml  
?categoryName=Overviews", true);
```

When XMLHttpRequest delivers a response, the response can either be interpreted as a DOM object or plain-text:

```
var domObject = xReq.responseXML  
alert(domObject) // Outputs "[Object]" (IE)  
// or "[object XML Document]" (Firefox)
```

```
var xmlString = xReq.responseText
alert(domObject) // Outputs the whole string
// i.e. "<category ...> ... </category>"
```

In most cases, you want to use `responseXML` because it gives you a well-structured DOM object: a hierarchical structure that can be navigated using industry-standard conventions. With recent browsers, you can rely on some basic parsing functionality to interrogate DOM objects. `HowToCreate` has a good list of DOM traversal functions (<http://www.howtocreate.co.uk/tutorials/texterise.php?dom=1>) and many of the Ajax Patterns demos perform such parsing (e.g., the Basic Wiki Demo; see <http://ajaxify.com/run/wiki>).

If you're parsing XML, consider delegating to a cross-browser library. *Sarissa* (<http://sarissa.sourceforge.net>), for example, has a variety of cross-browser XML manipulation routines. Another library, *Interactive Website Framework (IWF)* (<http://sourceforge.net/projects/iwff/>), simplifies XML expressions. Instead of the following standard DOM query:

```
var node = doc.documentElement.firstChild.firstChild.getAttribute("size");
```

you can write this:

```
var node = doc.groceries.frozen[0].pizza[0].size;
```

With a DOM object representing server data, the receiver can do one or more things:

- Transform the XML into some HTML. (See "Decisions," next.)
- Interrogate it to decide what to do next.
- Store it for later use in an *XML Data Island*.
- Modify it based on client state and upload it again.

This discussion has mostly focused on downloading XML from server to browser. That's the most common direction, but XML can also flow upwards too. It's sometimes easier to upload complex data as an XML document than as a set of CGI-style variables. To do this with `XMLHttpRequest`, you simply pass in the whole XML string as the argument to `send()`. You'll also need to ensure the transport type is suitable—typically, POST (or PUT).

```
xReq.send(xmlString);
```

The XML string being sent usually represents browser state or user input. Most often, it's simply built up with some manual coding, appending strings together and including variables where appropriate. However, you can also use a library like *Sarissa* or *Anders Noras's XmlSerializer* (<http://dotnetjunkies.com/WebLog/user/Profile.aspx?UserID=1095>) to convert from a DOM object—or some other object—into an XML string.

Finally, a note of caution. "XML" is the X in "AJAX," so it's sometimes considered a core Ajax technology. And its association with `XMLHttpRequest` also strengthens the argument for XML Messages. However, don't let all this make you feel obligated to use XML Messages. `XMLHttpRequest` supports transfers in any plain-text format, and it's fine to use *Plain-Text Messages* in many situations. XML is relatively simple, and there's good browser support,

but sometimes it just doesn't fit the bill. The "Solution" for Plain-Text Message compares the two approaches in more detail.

Decisions

How will the server generate XML?

There are many ways the server might generate XML messages:

- By using custom code that hand-creates the XML string.
- By building up a DOM object and serializing it.
- By using a generic framework to convert standard data structures into XML.
- By retrieving existing XML documents from the filesystem or external sources.

Will you specify a DTD or Schema?

When you pass XML Messages back and forth, each end must assume the same document format. That's a standard requirement with XML, and you can define the format precisely using a separate document: either a Document Type Definition or a stricter XML Schema document.

When the browser needs to render the message, how it will transform the XML?

The browser doesn't always need to render incoming XML messages—sometimes it just uses the data. But, for situations when it does render the XML, there are a few options. In all cases, the script is building up some HTML, which will then be injected onto a DOM element by setting its `innerHTML` property to the HTML string.

Manual JavaScript Conversion

For simple messages, you might find it easier to just perform a little parsing and manually create an HTML string.

XSLT

If you have the skill base and a suitable framework for the browsers you're targeting, *Browser-Side XSLT* allows for powerful conversion of XML.

Templating

Sometimes, *Browser-Side Templating* is a happy medium between the above two approaches. You still have to parse the XML document somehow, but the HTML generation is more straightforward.

Real-World Examples

Netflix Top 100

When you roll over a movie title in the Netflix Top 100 (<http://www.netflix.com/Top100#>), a balloon soon appears with summary details (Figure 9-9). After the rollover, an *XMLHttpRequest Call* occurs, which receives movie details like this:


```

<MOVIES>
<MOVIE ID="60031236" POS="17" DS="0">
  <TITLE>Kill Bill: Vol. 2</TITLE>
  <SYNOPSIS>In this film noir tale written ... </SYNOPSIS>
  <DETAILS RATED="R" RELYEAR="2003" GENREID="296" GENRENAME="Action && Adventure"/>
  <STARRING>
    <PERSON ID="92495" NAME="Uma Thurman"/>
    <PERSON ID="20008295" NAME="Lucy Liu"/>
  </STARRING>
  <DIRECTOR>
    <PERSON ID="20001496" NAME="Quentin Tarantino"/>
  </DIRECTOR>
</MOVIE>
</MOVIES>

```

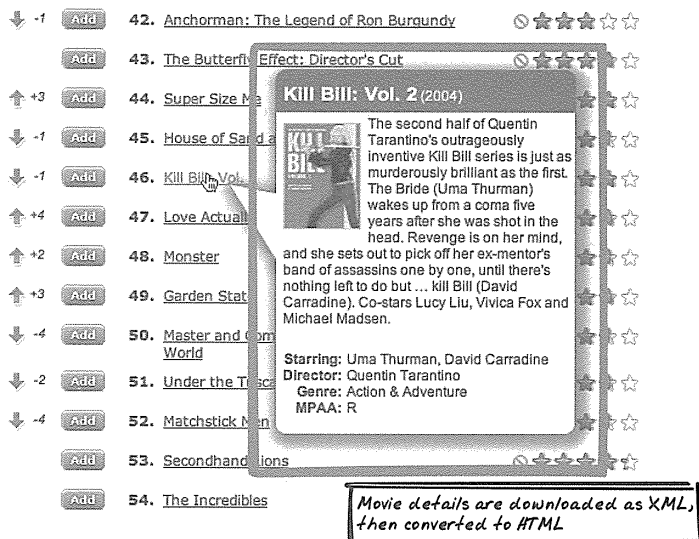


FIGURE 9-9. Netflix

Protopage

Protopage (<http://protopage.com>) is a portal-style application with excellent personalization capabilities. Each time you change something on the page, a command is uploaded to the server via an *XMLHttpRequest* Call. For example, here's what the browser sent when I moved a *Portlet* around:

```

<command accountId="25570" protopageId="25568" protopagePath="mahemoff" name=
"save-panel-geometry">
  <param name="id">129644</param>
  <param name="left">216</param>
  <param name="top">476</param>
  <param name="width">423</param>
  <param name="height">71</param>

```

```

    <param name="collapsed">false</param>
    <param name="zIndex">33</param>
    <param name="verticalScrollProportion">0</param>
  </command>

```

Google Maps

Google Maps is perhaps the most famous usage of XML Messages. Meta-information is downloaded as XML and rendered with *Browser-Side XSLT*.

Code Refactoring: AjaxPatterns XML Sum

The Basic Sum Demo (<http://ajaxify.com/run/sum>) uses a *Plain-Text Message* to transfer the sum result from server to browser. So if the browser sends a GET query requesting the sum of 4 and 8, the entire response is "12." That works fine, but sometimes we'd like a response to contain the original query too—it's convenient for caching, for instance (see *Browser-Side Cache* [Chapter 13]). If we're going to provide the original figures of the sum, the data's becoming a bit more complex—we now have a list as well as different types of data. To keep the format self-describing, let's refactor to XML.

The XML Sum Demo (<http://ajaxify.com/run/sum/xml>) behaves the same as the original version, but the server side returns results like this (<http://ajaxify.com/run/sum/xml/sumXML.php?figure1=4&figure2=8&figure3=>):

```

<sum>
  <inputs>
    <figure id="1">4</figure>
    <figure id="2">8</figure>
    <figure id="3"></figure>
  </inputs>
  <outputs>12</outputs>
</sum>

```

To avoid confusion, the server-side service is now called `sumXML.php`, and reflecting the new location is the only change required to the browser call. The server-side service has been altered to output the full XML. Note the XML content-type declaration in the header, which is necessary for the *XMLHttpRequest Call*.

```

<?
header("Content-Type: text/xml");

$sum = $_GET["figure1"] + $_GET["figure2"] + $_GET["figure3"];

echo <<< END_OF_FILE
<sum>
  <inputs>
    <figure id="1">{$_GET["figure1"]}</figure>
    <figure id="2">{$_GET["figure2"]}</figure>
    <figure id="3">{$_GET["figure3"]}</figure>
  </inputs>
  <outputs>$sum</outputs>
</sum>

```

```
END_OF_FILE
```

```
?>
```

In this refactoring, the figures aren't used, but the sum is still required. The callback function therefore navigates through the response XML Message to obtain the sum figure.

```
function onSumResponse(xml, headers, callingContext) {  
    var sum = xml.getElementsByTagName("outputs")[0].firstChild.nodeValue;  
    self.$("sum").innerHTML = sum;  
}
```

Alternatives

Plain-Text Message

As mentioned earlier in the "Solution," XML can often be overkill for simple messages, and *Plain-Text Messages* (see earlier in this chapter) are worth considering as an alternative.

JSON Message

Just like XML Message, *JSON Message* is a suitable representation for data of various complexities. The "Alternatives" section of JSON Message compares the two formats.

JSON Message



JSON, Marshal, Semantic, Serialize, YAML

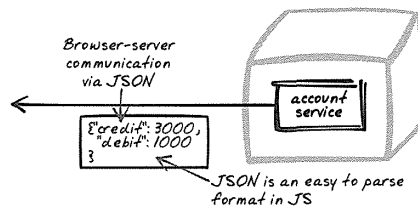


FIGURE 9-10. JSON Message

Developer Story

Devi is creating an Ajax calendar. The browser periodically polls for new appointments, which the server is sending as JSON messages. Since JSON messages are just JavaScript code for object creation, Devi's browser script needs only to run `eval` against each message in order to reconstruct the appointment.

Problem

How can you transfer data between server and browser?

Forces

- Ajax Apps require messages to be transmitted back and forth.
- Both browser and the server must be able to access the message. That usually means the format must be easily accessed in JavaScript as well as whichever server-side language is used.

Solution

Pass messages between server and browser in JavaScript Object Notation (JSON)

format. JSON is a standard serialization format, created in 2002 as a cleaner and lighter alternative to XML. As with XML, the object can range in complexity from a simple string to a deep hierarchical structure. Also like XML, JSON is language-neutral, meaning that you could marshal a C++ object into JSON notation and unmarshal it to form an object in Perl. But in practice, JSON is particularly suited to browser-server communication because it's a format based on JavaScript.

In fact, a JSON Message is JavaScript. This is a valid JSON Message:

```
"Homer J."
```

You can test browser-based JSON conversion on the Basic JSON Demo (<http://ajaxify.com/run/json/>). As you'll see there, the JSON Message "Homer J." maps to the JavaScript string, Homer J..

Here's a more complex example (as on the demo, reformatted):

```
{ "houseNumber": "742",  
  "street": "Evergreen Terrace",  
  "city": "Springfield",  
  "postcode": "49007",  
  "country": "USA",  
  "comments": ["Deliveries accepted.", "Familiar address, huh?", ""]  
}
```

As you can see, this JSON message is just a JavaScript object literal. You convert it like this:

```
var name = eval("(" + nameJSON + ")");
```

Or like this (<http://jibbering.com/2002/4/httprequest.html>) if it takes your fancy:

```
var name=new Function("return " + nameJSON)();
```

Note that you don't even need a JSON library in the browser. The browser can pick up a JSON string using an *XMLHttpRequest Call* and simply run the standard eval function.

However, there is actually a JavaScript JSON library (<http://www.crockford.com/JSON/js.html>), which adds two important capabilities: `parse()` for safer string-to-object conversion, and `stringify()` for object-to-string conversion. The former is an alternative to manual JavaScript evaluation, because evaling arbitrary messages puts your app at risk of running

malicious JavaScript code. Thus, the string-to-object conversion should be executed if you don't trust the message. To parse a string, use the following code:

```
var name = JSON.parse(nameJSON);
```

If you need to upload something to the server, you'll need to convert a JavaScript object to JSON with `stringify()`. The Basic JSON Demo (<http://ajaxlocal/run/json/>) shows this conversion process too. The call looks like this:

```
var nameJSON = JSON.stringify(name);
```

So far, the examples have considered only JavaScript conversion. But JSON wouldn't be very useful if you could only convert to and from JavaScript objects—you need to convert at the other end too, and your server side is probably not written in JavaScript. That's why there are JSON processors for many languages. Using these processors, you can easily share an object between JavaScript and your favorite server-side language.

A remoting modification (<http://ajaxify.com/run/json/remoting>) of the Basic JSON Demo (<http://ajaxify.com/run/json/>) sends JSON to the server, using *XMLHttpRequest Calls*. There, it's converted to standard PHP objects using Michael Migurski's JSON-PHP library (<http://mike.tecno.com/json.html>). The library works similarly to JSON libraries for Java, .Net, and other languages. The following code converts JSON to a standard object:

```
$json = new JSON();  
$newObject = $json->decode($jsonString);
```

while the following code performs the reverse operation:

```
$json = new JSON();  
$json = $json->encode($object);
```

Real-World Examples

Many web sites use JSON as their data transfer format for shuttling data between browser and server.

Kiko

Kiko (<http://kiko.com>) is an online calendar application with a slew of Ajax features (Figure 9-11). As you'd expect, the server holds a model of the calendar, and the browser keeps uploading changes using *XMLHttpRequest Calls*. JSON is the message format used for server responses—each response is a list of objects.

Delicious JSON Feed

Delicious (<http://del.icio.us/doc/feeds/json/>), a social bookmarking tool, provides a *Web Service* that exposes a user's recent bookmarks in the form of a JSON Message. For reasons discussed in *On-Demand JavaScript* (Chapter 6), this means a browser script can conveniently grab the data without the need for a *Cross-Domain Proxy*.

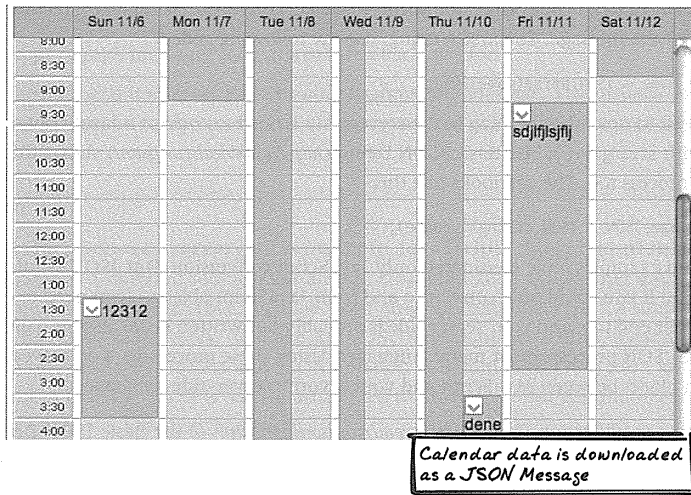


FIGURE 9-11. Kiko

Route Planning

Jim Ley's Route Planning application (<http://jibbering.com/routeplanner/>) shows you all the routes for a given airport (Figure 9-12), and, as his long-running XMLHttpRequest Tutorial (<http://jibbering.com/2002/4/httprequest.html>) explains, it uses JSON. For example, a JSON Message for the LAX airport is available at <http://jibbering.com/routeplanner/route.1?LAX>. What the browser receives from the following JSON Message is an *XMLHttpRequest Call* (... has been substituted for multiple data items):

```
{from: 'LAX', airports: [{id: "AMS", country: "NL", lat: "52.316666666667", lon: "4.7833333333333", tz: "Europe/Amsterdam", name: "Amsterdam", shortcode: "Schiphol"}, ..., {id: "IAD", country: "US", lat: "38.95", lon: "-77.45", tz: "America/New York", name: "Washington, DC", shortcode: "Washington Dulles International"}], routes: [{carrier: "star", toAirport: "AMS", miles: 5570}, ..., {carrier: "oneworld", toAirport: "DCA", miles: 2304}]}
```

Ajax.Net framework

Ajax.Net (<http://ajax.schwarz-interactive.de/>) is one of several *Ajax Stub* frameworks that uses JSON Messages to transfer data, which can easily be converted to and from native objects at either end. For more details, see "JSON Message" in the "Related Patterns" section of *Ajax Stub*.

Code Example: Kiko

Kiko (<http://kiko.com>) responds to *XMLHttpRequest Call* with JSON Messages. When you log in, it downloads a list of appointments like this:

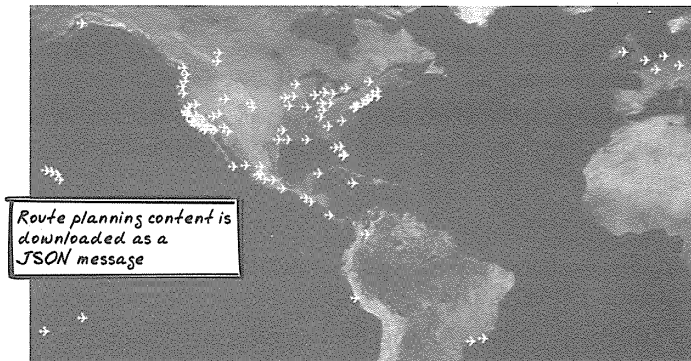


FIGURE 9-12. Jibbering Route Planner

```
[
  {
    "title": "text",
    "isevent": "bool",
    "picture": "text",
    "starttime":
      "timestamp",
    "endtime": "timestamp",
    "recurs": "int2",
    "recurend": "timestamp",
    "insystem":
      "timestamp",
    "recurstart": "timestamp",
    "recurweek": "int2",
    "description":
      "text",
    "defaultfree": "bool",
    "location": "text",
    "apptid": "int8"
  },
  {
    "title": "Roundup
meeting",
    "isevent": "t",
    "picture": "",
    "starttime": "2005-09-16 19:00:00",
    "endtime": "2005-
09-16 19:30:00",
    "recurs": "1",
    "recurend": "",
    "insystem": "2005-09-12 14:52:10.
965713",
    "recurstart": "2005-09-16 04:00:00",
    "recurweek": "16",
    "description": "Hopefully
just a quick roundup",
    "defaultfree": "f",
    "location": "Cyberspace",
    "apptid": "9222"
  },
  {
    "title": "Go Home!",
    "isevent": "t",
    "picture": "",
    "starttime": "2005-09-15 21:00:
00",
    "endtime": "2005-09-15 21:30:00",
    "recurs": "0",
    "recurend": "",
    "insystem": "2005-09-12
15:00:33.659793",
    "recurstart": "",
    "recurweek": "",
    "description": "",
    "defaultfree":
      "f",
    "location": "Office, Bar, Home",
    "apptid": "9288"
  }
]
```

Note that the message format here is a little unusual: everything is Strings, whereas JSON can store booleans and numbers directly. Consequently, the first element of the array includes some metadata to facilitate conversion, and the JavaScript includes a generic function to create native JavaScript objects from JSON messages like this.

Kiko's browser script also includes some conversion to JSON, as shown below (though I was unable to exercise this code). Typically, requests in Kiko are made as POSTs with CGI-style parameters, and only the responses are JSON.

```
var obj=json.stringify({'type':'newuser','email':email})
```

Alternatives

XML Message

JSON defines itself as a “fat-free alternative to XML.” Before considering the differences between the formats, let's first look at what they have in common.

- Both are formats that depict an object as a plain-text string.
- Being plain-text string formats, both are suitable for transfer across HTTP. This means that both are suitable as inputs and outputs for a *Web Service*.
- Each format is supported by libraries in numerous languages, including JavaScript. There are libraries to convert from native objects to either format and back again to native objects.

JSON Message has several advantages over *XML Message* (see earlier in this chapter):

- JSON is more compact, and the lack of tags sometimes leads to a better visual representation of the underlying data.
- It's often claimed that JSON is faster for browsers to parse, though a recent investigation (<http://blogs.ebusiness-apps.com/dave/?p=45>) suggests XML parsing does scale better.
- JSON is a concrete data format. XML, in contrast, is really a meta-format, and a developer has many choices to make about the precise XML dialect to use. Each mapping strategy has its own conventions—for example, developers need to decide on tag names and between tag attributes or nested tags. Consequently, a server-side XML-object mapping framework may not be message-compatible with a JavaScript counterpart. Also, don't underestimate the amount of meetings and emails that will be necessary to resolve a seemingly trivial argument over data formats. In that sense, JSON is closer to a defined XML dialect than to XML itself.
- Within the browser, JSON has the edge in terms of support and consistency, since it's based on standard JavaScript.
- JSON happens to be quite compatible with YAML (“YAML Ain't Markup Language”; see <http://www.yaml.org/about.html>), a similar lightweight alternative to XML that's gaining traction in the dynamic scripting community.

XML Message (see earlier in this chapter) has several advantages over JSON Message:

- XML is vastly more familiar to the IT community than JSON.
- XML is more self-documenting. The header identifies which XML format is being used, and there's often a schema or DTD that defines the format precisely.
- XML has much more support in terms of libraries and tool support. JSON libraries tend to be simply about conversion, which its advocates would probably argue is all that's required. XML, on the other hand, has support in terms of DTD and Schema validators, XPath tools to interrogate the data, XSLT processors to perform translations, and so on. Furthermore, many IDEs, editors, and debugging environments make XML easy to work with.
- For any given task, developers usually have the luxury of choosing between several competing implementations.

Related Patterns

Ajax Stub

Being a portable object format, JSON is a useful way to facilitate calls from browser to server using an *Ajax Stub* (see earlier in this chapter) framework.

On-Demand JavaScript

Since a JSON Message is an ordinary JavaScript expression, it can be pulled in using *On-Demand JavaScript*. Data from external domains can be accessed this way, as explained in On-Demand JavaScript (Chapter 6).

Browser-Server Dialogue

WE'VE LOOKED AT THE BASIC TECHNOLOGIES FOR WEB REMOTING, WHICH LEADS TO QUESTIONS ABOUT browser-server information flow. How will you deal with multiple calls? How will you keep the browser and the server synchronized? How will you access external domains?

Call Tracking is about tracking calls and dealing with the asynchronous nature of Ajax. The next three patterns concern synchronization. With *Periodic Refresh*, the browser keeps requesting fresh information from the server. The opposite is *Submission Throttling*, where the browser keeps uploading new information to the server. An alternative to Submission Throttling, *Explicit Submission* involves uploading only when the user performs some action.

All of those patterns help manage bandwidth, but keeping things in sync can still be quite complex when there are lots of entities in the server and a rich interface in the browser. *Distributed Events* help manage the complexity.

Finally, *Cross-Domain Proxy* is a technique for mediating the dialogue between the browser and external servers.

Call Tracking



Asynchronous, Follow, Monitor, Parallel, Tracking, XMLHttpRequest

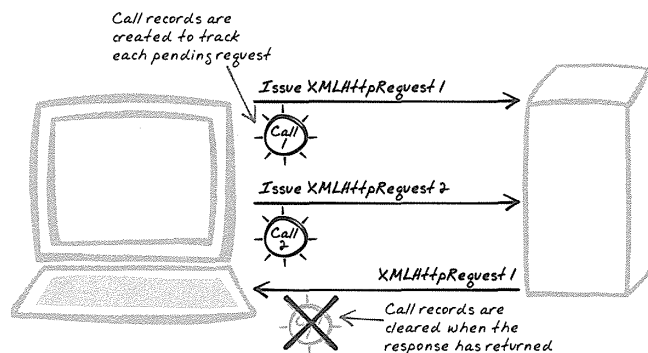


FIGURE 10-1. Call Tracking

Developer Story

Dave is writing an Ajax chat client, and he's concerned the more vocal users will hog server resources, so he decides to enforce a policy of no more than three pending server calls at any time. Thus, the browser script pushes messages into a queue and tracks the progress of each *XMLHttpRequest Call* (Chapter 6) containing those commands. He can enforce his maximum call count policy by tracking the state of each call.

Problem

How can you control parallel *XMLHttpRequest Calls* (Chapter 6)?

Forces

- Fast users and busy applications often stretch networks and servers to their limits.
- Asynchronous interaction is the only practical approach for Ajax applications. You don't want to block all interaction just because the network happens to be slow.
- The number of simultaneous requests must be controlled, because browsers can only handle a few at a time, and also to reduce the overall load.

Solution

Track XMLHttpRequest calls as they proceed from browser to server and back again. XMLHttpRequest is a fairly basic component that needs to be augmented or wrapped for better control over asynchronous dialogue. Furthermore, it is useful to keep all these wrappers in a collection. Note that this pattern is fairly low level, and the details should generally be encapsulated in a wrapper library.

The standard mechanism for Call Tracking requires an XMLHttpRequest wrapper. Consider the implementation of the Ajax Client Engine (ACE) library (<http://www.lishen.name/>). A Requester abstraction creates an XMLHttpRequest object upon construction:

```
function Requester()
{
    var requester;
    if (window.XMLHttpRequest)
    {
        requester = new window.XMLHttpRequest();
        ...
    }
    ...
}
```

Requester's response handler doesn't go directly back to the caller's registered event handler, but instead to a method of Requester. This internal callback can then perform logging and other tasks before passing control back to the caller's handler. There's also a caching mechanism, so the Requester will add and remove itself from the cache as the call progresses.

The most important reason to track calls is to tame the asynchronous nature of XMLHttpRequest. There's a potential bug in which the programmer treats XMLHttpRequest as a Singleton (Gamma et al., 1995). That is, there's only a single, global, XMLHttpRequest instance, but parallel calls are made (<http://smokey.rhs.com/web/blog/PowerOfTheSchwartz.nsf/d6plinks/RSCZ-6CDPEX>). An initial call will be pending, when suddenly a new call steps in. What happens here is unclear, as it's the sort of unanticipated behavior that will vary across browser implementations, but it certainly won't be a happy result. The first call will possibly be cancelled, and it's unlikely its event handler will be notified. The simplest way to resolve this problem is to create a new XMLHttpRequest instance each time, then use a closure; i.e.:

```
xhReq.onreadystatechange = function() {
    if (xhReq.readyState != 4) { return; }
    var serverResponse = xhReq.responseText;
    ...
};
```

We can consider this a limited form of Call Tracking, since we've at least ensured each pending call will have its own associated XMLHttpRequest. Still, we can achieve a lot more by wrapping requests and placing the wrappers in a collection, as demonstrated by ACE and other libraries:

Pooling

A collection of wrappers can be used as a cache. When each object has delivered its response, it's returned to the available pool (and its state reset in case of an error). This reduces object creation and destruction.

Limiting

Using a fixed-size pool (see the previous point), you have the option of limiting the number of simultaneous requests to reduce network load and prevent problems in the browser; most browsers will only allow a handful of pending requests at any time.

Detecting timeouts

As explained in *XMLHttpRequest Call* (Chapter 6), the framework can start a timer when the request is issued and notify the caller if a timeout occurs.

Sequencing

With all requests coordinated by a central piece of logic, it's possible to influence the sequence of calls going in and out of the server. While it's best to design for calls to be completely parallel, sometimes you might have constraints; e.g., you might want the server to process calls in the same order as the user initiates them; i.e., fire a request only when the previous response has been received.

Atomic Processing

Related to the previous point, you often need to ensure the browser will handle one response completely before moving onto the next. One scenario is appending a bunch of response information—if the responses are dealt with in parallel, you'll end up with interleaved messages. One style of Call Tracking is to create JavaScript Command (Gamma et al.) objects from each response, push them onto a queue, and have a separate thread execute them one at a time.

Passing in Call Context

Sometimes, a service's response is minimal and doesn't include any detail about the original call—for example, `false` as opposed to `<spellcheck term="misspeld">false</spellcheck>`. Indeed, sometimes there's no response at all, and the framework will need to inform the response handler of a timeout. In these cases, it's useful to provide some context about the original call. By tracking the call, a framework can remember a "call context" that the caller sets when the request is issued. When the framework eventually transmits the response to the caller's response handler, it also passes in the call context. See "Code Refactoring: AjaxPatterns Predictive Fetch Sum" in *Predictive Fetch* (Chapter 13) to see how a calling context can be used.

Logging

A wrapper can register itself as the wrapped object's response handler, and then log any changes. It can also poll for any new content (since there's no guarantee the event handler will be called when new content is added).

Real-World Examples

Ajax Client Engine (ACE) library

Li Shen's Ajax Client Engine (ACE) (<http://www.lishen.name/>) uses Call Tracking to ease development and harden the application in production. Among its many features are several related to Call Tracking:

- Long calls are timed out.
- Changes to XMLHttpRequest's response state are logged.
- A service can be periodically polled.
- The caller can declare exactly when the callback method should be invoked.

AjaxCaller library

The AjaxCaller library (<http://ajaxify.com/run/Lib/js/ajaxCaller.js>), used throughout the AjaxPatterns demos for Web Remoting, uses Call Tracking to pool calls, which wrap XMLHttpRequest objects.

libXmlRequest library

libXmlRequest (<http://www.whitefrost.com/reference/2005/09/09/libXmlRequest.html>) is another XMLHttpRequest wrapper. It keeps XMLHttpRequest objects in a pool so they can be reused, and tracks the response status in order to manage the pool.

Code Example: Ajax Client Engine (ACE)

In this example, we'll look at ACE's internal handling of call timeouts. The library consists of Requester objects, which wrap XMLHttpRequest objects. As explained earlier in the Solution, the wrapped XMLHttpRequest objects are created upon construction. When the wrapper is invoked to make a call, it creates a timer to cancel the request if it takes too long. The timer ID is held as an attribute of the wrapper.

```
timeoutId = window.setTimeout(endRequest, request.callbackTimeout * 1000);
```

To track the call, the wrapper registers itself as a request handler:

```
function beginRequest()
{
    ...
    requester.onreadystatechange = readystatechangeHandler;
    ...
}
```

Its handler is therefore called upon each change, and in the case of a complete call, cancels the timer:

```
function readystatechangeHandler()
{
    ...
    if (requester.readyState == Ace.ReadyState.Complete)
    {
        ...
        if (requester.status == Ace.Status.OK)
        {
```

```

    ...
    if (timeoutId)
    {
        window.clearTimeout(timeoutId);
        timeoutId = undefined;
    }
    ...
}
...
}

```

Alternatives

Fire-and-forget

Some calls need no tracking because they are low-priority uploads of information to the server. For example, a chat app might reasonably ignore the results of uploading the user's messages because a separate thread is continuously polling for *all* recent messages. Another example would be uploading information to support *Predictive Fetch* (Chapter 13), where the worst case is simply the lost opportunity of a performance optimization. Ajaxian.com featured an interesting article on optimizing for this kind of request (http://www.ajaxian.com/archives/2005/09/ajaxian_fire_an.html).

Global XMLHttpRequest

You may be able to get away with a single, global XMLHttpRequest under certain conditions and with care. For example, you can use a lock to ensure there's only one pending call at each time (which is really a special case of Call Tracking). However, you risk forcing the user to endure long waiting periods.

Metaphor

Call Tracking is like tagging self-addressed envelopes before you send them away, so you can track them as they return.

Acknowledgments

This pattern was originally inspired by Richard Schwartz's caution against the familiar anti-pattern of working with a global XMLHttpRequest (<http://smokey.rhs.com/web/blog/PowerOfTheSchwartz.nsf/d6plinks/RSCZ-6CDPEX>).

Periodic Refresh



Auto-Update, Polling, Sync, Synchronise, Synchronize, Real-Time

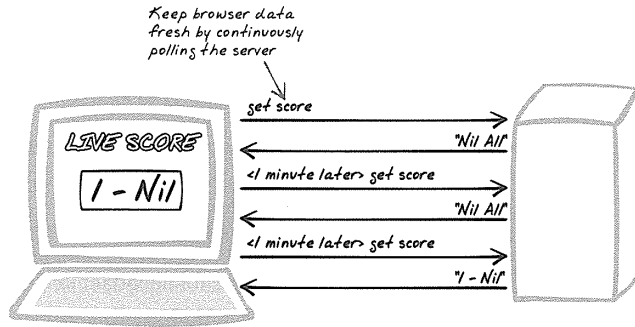


FIGURE 10-2. Periodic Refresh

Developer Story

Devi's coding up a ticket sales web site. For each event, she wants to keep the browser updated with the number of tickets remaining. Thus, she introduces a timer so that every 30 seconds, it calls a web service to pull down the latest sales stats.

Problem

How can the application keep users informed of changes occurring on the server?

Forces

- The state of many web apps is inherently volatile. Changes can come from numerous sources, such as other users, external news and data, results of complex calculations, and triggers based on the current time and date.
- HTTP requests can only emerge from the client. When a state change occurs, there's no way for a server to open connections to interested clients.
- One way to keep the browser updated is *HTTP Streaming* (Chapter 6), but, as the "Alternatives" section for that pattern explains, it's not always ideal. In particular, it's not very scalable.

Solution

The browser periodically issues an `XMLHttpRequest` Call to gain new information; e.g., one call every five seconds. The solution makes use of the browser's *Scheduling* (Chapter 7) capabilities to provide a means of keeping the user informed of latest changes.

In its simplest form, a loop can be established to run the refresh indefinitely, by continuously issuing *XMLHttpRequest Calls* (Chapter 6):

```
setInterval(callServer, REFRESH_PERIOD_MILLIS);
```

Here, the `callServer` function will invoke the server, having registered a callback function to get the new information. That callback function will be responsible for updating the DOM according to the server's latest report. Conventional web apps, even most of those using *XMLHttpRequest Calls*, operate under a paradigm of one-way communication: the client can initiate communication with the server, but not vice versa. Periodic Refresh fakes a back-channel: it approximates a situation where the server pushes data to the client, so the server can effectively receive new information from the browser. Indeed, as some of the examples show, the server can also mediate between users in almost real-time. So Periodic Refresh can be used for peer-to-peer communication too.

But before we get too carried away with Periodic Refresh, it's important to note that it's a serious compromise, for two key reasons:

- The period between refreshes would ideally be zero, meaning instant updates. But that's not realistic and the browser will always lag behind. Latency is particularly problematic when the user is interacting with a representation of volatile server-side data. For instance, a user might be editing an object without knowing that another user has already deleted it.
- There is a significant cost attached to Periodic Refresh. Each request, no matter how tiny, demands resources at both ends, all the way down to operating-system level. Traffic-wise, each request also entails some bandwidth cost, which can add up if refreshes are occurring once every few seconds.

So a key design objective must be to increase the average refresh period and reduce the content per refresh, while maintaining a happy user experience. One optimization is a timeout: the refreshes cease when the system detects the user is no longer active, according to a *Timeout* (Chapter 17) mechanism. You also want to make sure each refresh counts; it's wasteful to demand lots of updates if the network isn't capable of delivering them. Thus, the browser script can do some monitoring and dynamically adjust the period so it's at least long enough to cope with all incoming updates. Many of the Performance Optimization patterns can also be applied to Periodic Refresh—see the next section.

Decisions

How long will the refresh period be?

The refresh period can differ widely, depending on usage context. Broadly speaking, we can identify three categories of activity level:

Real-Time interaction (milliseconds)

The user is actively interacting with the system, and his input relies on the server's output—for example, a chat user needs to see what others are saying, a trader needs to see

current prices, or a game player needs to see the state of the game. Here, the interval could be as low as a millisecond on a local machine or local network, or perhaps be 20–100 milliseconds on a global network.

Active monitoring (seconds)

The user relies on the server state for work outside the system—for example, a security officer watches sensor displays for suspicious activity, or a manager occasionally watches the fluctuation of sales figures during a particular window of time. In some cases, timeliness may be critical, making subsecond responses desirable. In other cases, a few seconds is sufficient feedback.

Casual monitoring (minutes)

Some applications are designed for the user to leave in a window or separate browser tab and view throughout the day. The information does not change often, and it's no drama if the user finds out a little later. Prime candidates here are portals and RSS aggregators. Refresh periods of 10 minutes or more are often acceptable for such content. A manual "Refresh Now" mechanism is worth including where the refresh period is longer than a few seconds. It can relate to the entire application or to specific components.

Sometimes, the best solution uses multiple Periodic Refresh cycles in parallel, each with a frequency reflecting the user's needs. An interactive wiki, for example, might update news headlines every 10 minutes, online statuses of other users every minute, and content being edited every second.

Real-World Examples

Lace Chat

Instant messaging (or "online chat") applications pre-date the Web, and, unlike email and other services, web interfaces have never quite worked out, partly due to the fact that people don't enjoy frequent full-page refreshes. Ajax makes it possible to avoid a complete refresh by pulling down messages with an *XMLHttpRequest Call* (Chapter 6). Only the new messages need to be sent in each periodic response. In fact, there are several applications under development. Lace Chat (<http://socket7.net/lace/>), for example, is only a proof-of-concept, but provides good evidence that web-based chat is feasible. Every few seconds the messages update to show any new messages other users may have entered. When you post a message, it's also handled as an *XMLHttpRequest Call*.

Magnetic Poetry

Like a wiki, Magnetic Poetry (<http://www.broken-notebook.com/magnetic/>) involves a shared workspace (Figure 10-3). In this case, users move tiles through the space, and the application updates once a second to reflect the new space. As of version 1.7, the entire tile set is sent each time, but there's the potential to compress the information by sending only recent tile positions. This enables two users to work on the area simultaneously, and one can even see a tile being dragged by a different user, like a low-frequency animation.

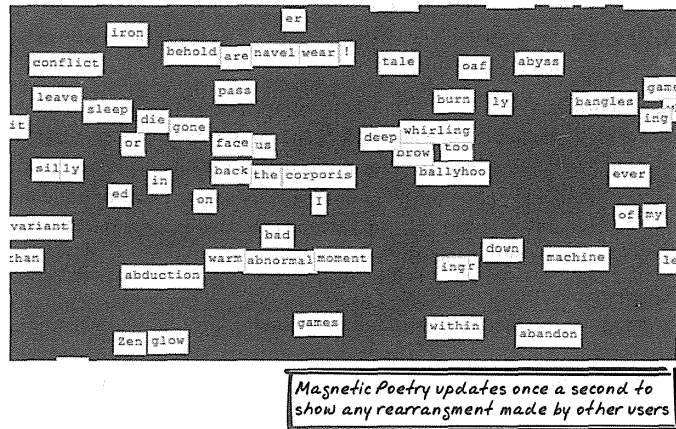


FIGURE 10-3. Magnetic Poetry

Claude Hussenet's portal

Portals, by definition, display various kinds of information. And often that information is of a dynamic nature, requiring periodic updates. Claude Hussenet's portal (<http://www.claudehussenet.com/portal/Welcom.do>) contains several portlets:

- World news headlines (taken from Moreover—see <http://moreover.com>—and Yahoo! News—see <http://news.yahoo.com>). The server appears to generate these from an RSS feed.
- New online articles appearing on TheServerSide.com and DevX.com. Again, these are taken from RSS feeds.
- Customized stock portal. The server appears to maintain a record of all current stock prices and is capable of delivering those requested by the browser.

In each case, the information is volatile and needs to be periodically updated, as is typical for many portlets. Also characteristic of portal applications is the relatively long refresh period, 15 minutes in this case. Each portlet contains a manual refresh too, for immediate results.

Code Examples

Lace

Lace Chat (<http://socket7.net/lace/>) handles Periodic Refreshes to show all users' chat messages.

The timer is set on startup to periodically call the `get()` function, which initiates the *XMLHttpRequest Call* (Chapter 6) for new messages:

```
this.timerID = setInterval(function () { thisObj.get(true); }, interval);
```

get() performs a straightforward query of the server:

```
this.httpGetObj.open('POST', this.url, true);
this.httpGetObj.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded; charset=UTF-8');

var thisObj = this;
this.httpGetObj.onreadystatechange = function () { thisObj.handleGet(system); };
this.httpGetObj.send(param);
```

If the server has changed at all, the entire contents are returned. How does Lace know if the server has changed? Each time the server responds, it generates a hash for the contents. And when the browser next calls for a refresh, it includes the last hash as a parameter to the call. The server sends a full response only if the current hash differs from that specified by the browser:

```
[lib_lace.php]
function getMessageHash() {
    ...
    return md5(implode(file(LACE_FILE)));
}

[lace.php]
$_hash = getMessageHash(); if ($_hash == $hash) exit; // no change exit($_hash.
'||||'.getFileContentsRaw());
```

Code Refactoring: AjaxPatterns Periodic Time

The Basic Time Demo (<http://ajaxify.com/run/time>) requires the user to manually update the time display by clicking a button. We'll perform a quick refactoring to re-fetch the time from the server every five seconds (<http://ajaxify.com/run/time/periodicRefresh>).

First, we'll create the function that initiates the server calls. Here, we already have two functions like that, one for each display. So, let's ensure we call both of those periodically:

```
function requestBothTimes() { requestDefaultTime(); requestCustomTime(); }
```

Then, the Periodic Refresh is simply a matter of running the request every five seconds:

```
function onLoad() { ... setInterval(requestBothTimes, 5000); ... }
```

One more nicety: the function in setTimeout begins to run only after the initial delay period. So we're left with empty time displays for five seconds. To rectify that, requestBothTimes is also called on startup:

```
function onLoad() { ... requestBothTimes();
                    setInterval(requestBothTimes, 5000); ... }
```

Now, the user sees the time almost as soon as the page is loaded.

Alternatives

HTTP Streaming

One of the forces for this pattern is that HTTP connections tend to be short-lived. That's a tendency, not a requirement. As *HTTP Streaming* (Chapter 6) explains, there are some circumstances where it's actually feasible to leave the connection open. Streaming allows for a sequence of messages to be downloaded into the browser without the need for explicit polling. (Low-level polling still happens at the operating system and network levels, but that doesn't have to be handled by the web developer, and there's no overhead of starting and stopping an HTTP connection for each refresh, nor of starting and stopping the web service.)

Related Patterns

Distributed Events

You can use *Distributed Events* (see later in this chapter) to coordinate browser activity following a response.

Fat Client, Browser-Side Cache, Guesstimate

A little work on performance issues can help make the system feel more responsive. Some of the performance optimization patterns help:

Fat Client (Chapter 13)

Reduces the need for server processing by pushing functionality into the browser.

Browser-Side Cache (Chapter 13)

Reduces queries by retaining query data locally.

Guesstimate (Chapter 13)

Gives the user a sense of what's happening without actually performing any query.

Submission Throttling

Submission Throttling (see the next pattern) also involves a periodic call to the server. The emphasis there is on uploading browser-side changes, whereas the present pattern focuses on downloading server-side changes. The two may be combined to form a general-purpose "synchronize" operation, as long as the update frequency is sufficient in both directions. This would improve performance by reducing the amount of overall traffic. The Wiki Demo (<http://ajaxify.com/run/wiki>) takes this approach.

Heartbeat

It's inevitable that users will leave their browser pointing at web sites they're not actually using. The rising popularity of tabbed browsing—now supported by all the major browsers—only exacerbates the problem. You don't want to keep refreshing the page for idle users, so use *Heartbeat* (Chapter 17) to detect whether the user is still paying attention.

Metaphor

A movie is refreshed at subsecond intervals to provide the illusion of real-time activity.

Submission Throttling



Buffer, Queue, Performance, Throttle

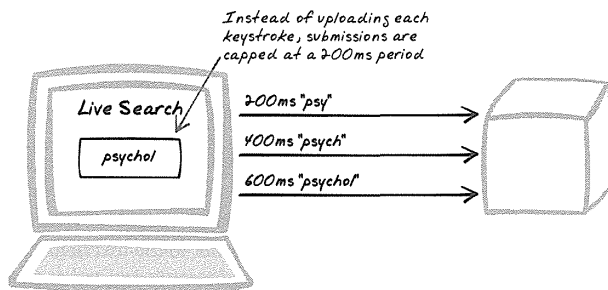


FIGURE 10-4. Submission Throttling

Developer Story

Devi's producing an Ajax chat web site and wants to transmit text as the user types. However, she knows some users will type faster than the system can cope with so she introduces a throttling mechanism that ensures no more than one message is uploaded every 200 milliseconds.

Problem

How can information be submitted to the server?

Forces

- Information is often uploaded in bursts; e.g., a chat tool incurs many hits when a user becomes passionate about the topic, or a data entry tool incurs many hits when the user responds to some new information.
- It's difficult for the server to cope with lots of messages at once.
- Browsers can only handle a limited number of pending *XMLHttpRequest Calls* at any moment.
- Each message has overheads, such as packet headers and requires some processing at each stage of the browser/server round-trip.

Solution

Instead of submitting upon each JavaScript event, retain data in a browser-based buffer and automatically upload it at fixed intervals. As with many applications of buffering and throttling, the purpose is to strike a balance between responsiveness and resources. In most cases, it would be ideal to respond to every keystroke and mouse movement—so, for example, like a desktop application, tooltips could come directly from the server as the user mouses around. But that’s not practical due to bandwidth considerations, and possibly server constraints too. So to ease bandwidth—and to lessen server processing, call detail is accumulated and periodically uploaded.

Exactly what the buffer holds is application-specific, but there are two general styles: data buffers and command queues.

In the first buffer style, the buffer holds some data to be uploaded. Consider implementing a *Suggestion* (Chapter 14) interface like Google Suggest (<http://www.google.com/webhp?complete=1&hl=en>), which keeps showing information based on what’s been typed. The simplest thing would be to add an “onchange” listener to the text field and upload the buffer when each change occurs. However, what if you get a fast typist, one of those folks who revels in their “words per minute” metric? Banging out 100 words per minute means perhaps 10 characters per second, or a call every 100 milliseconds—feasible with a local-host web server, maybe workable on an intranet, but not scalable for most public Internet applications. So *Suggestion* systems, and more generally *Live Command-Lines* (Chapter 14), run a fixed-period timer. Every 500 milliseconds, say, the browser checks if there was a change since the last call, and if so, uploads to get some information and remembers what was uploaded to avoid doing so again. Effectively, the result for the current text field is cached, and the cache can only change on fixed periods.

A similar pattern might be used on a *Live Form* (Chapter 14), where the entire form is periodically uploaded to the server, even though some fields are blank, along with an indication of the user’s progress. The server can then use some intelligence to critique the form data, so as to provide live feedback.

The other style of buffer is a command queue. Here, Commands (see Gamma et al., 1995) are held in a queue, and the whole queue periodically uploaded to the server before being cleared. Some kind of serialization must take place, so the Commands must be represented as Strings and the Queue must ensure they can be pulled apart by the server; e.g., by separating them with a delimiter character. It’s up to the developers to agree on a protocol for defining how Commands are represented as Strings. For example, a wiki might use the string “Del Issues” to delete the Issues page. Another technique would be to store the commands as custom JavaScript objects and serialize them into *JSON Messages* (Chapter 9).

Submission Throttling might appear to optimize technical resources at the expense of usability, but can actually be a boon for users too. There’s a reason why the Windows and Apple operating systems don’t show copious logging details when booting. Technical users

may appreciate the comprehensive output of a Linux boot sequence, but most users consider it overload—much more information than they actually care about. Throttling is a good way to prevent information overload in the browser, especially error messages that might come back because the user is only halfway through an edit.

Decisions

How will the server deal with incoming commands? Will all commands still be valid?

Submission Throttling is vulnerable to integrity issues, because synchronization is being deliberately downgraded. The universe will have moved on since the user submitted the original commands—the time will be different, new information may be available, existing information may have changed or been deleted, and other users may have executed commands in the meantime. All of these scenarios are unavoidable manifestations of the asynchronous nature of Ajax, and the design must take them into account.

The server needs to decide whether to process each incoming command as it's quite possible some are no longer valid. For example, a stock purchase order might be refused on the basis that the price has just risen or because the server has since expired an initial quote it made to the client. Some of these decisions can be made by standard techniques such as atomic database transactions, but others might require some custom business logic. Furthermore, some business logic will be required to decide what to do with the rest of the Commands in a queue, should one Command in the middle fail. Sometimes, it's okay to keep processing the rest and sometimes not.

How will buffer uploads be triggered?

The most obvious algorithm for upload sequencing is a timer. Every minute (say), the browser polls the buffer and makes the corresponding call sequence. Or, if nothing has changed, it might do nothing for another minute. If a timer is used, a decision must be made as to when it will trigger. First, the period is usually fixed, but does not have to be. It might be increased during times of known server activity, or even altered to respond to those constraints dynamically. Furthermore, it might be based on the user in question: service level can be tweaked by giving premium users shorter throttle periods.

A variant, usually more useful, is to cap the rate but immediately send the first command after a long wait. This helps a user who changes information only occasionally. Let's say the throttle period is 30 seconds. With a standard, fixed-interval, algorithm, the following sequence occurs:

1. 00 secs: system polls, no activity so no upload
2. 30 secs: system polls, no activity so no upload
3. 60 secs: system polls, no activity so no upload
4. 65 secs: infrequent user does something; no upload yet

5. 75 secs: infrequent user does something; still no upload
6. 90 secs: system polls and uploads both pending commands

We can still cap the rate at 30 seconds, but upload a command immediately if there's been no activity for the past 30 seconds:

1. 00 secs: system polls, no activity so no upload
2. 30 secs: system polls, no activity so no upload
3. 60 secs: system polls, no activity so no upload
4. 65 secs: infrequent user does something
5. 65 secs: system notices and uploads command immediately
6. 75 secs: infrequent user does something; this time, the command must wait since there was a recent call
7. 95 secs: system polls, uploads if there was any further activity after 65 secs

Prioritization is another technique. A timer might be used for regular events, but with priority given to any critical commands. A *Live Form* (Chapter 14) might periodically upload the progress of an individual field, so the server can provide suggestions, for example. But as soon as the user proceeds to the next field, a call takes place immediately.

A further consideration is the user's activity during upload. It's wise to avoid uploading something the user is currently working on, especially if other users will see it and if they probably won't be working on it much longer. So, one policy might involve uploading only after a period of idle activity.

How many buffers per browser application?

There's no reason to have just one buffer for all commands. It's possible to have several buffers running in parallel, providing you have considered the consequences of commands arriving in a different order to the user requesting them. Prioritization was already mentioned above, which would be one reason to have several buffers—higher-priority buffers being processed with greater frequency.

Here's how a blog reader might use three buffers in parallel:

- A low priority queue uploads comments submitted by the user. *Throttle period: 15 seconds.*
- A medium priority queue pulls down requested feeds from the server. *Throttle period: 3 seconds.*
- A high priority text buffer lets the user tag articles with a *Suggestion* mechanism. While it's not necessary to upload upon each keystroke, any new information must be transferred quite rapidly. *Throttle period: 0.2 seconds.*

This example illustrates that it's quite easy to run several buffers without threat to data integrity and without user confusion.

How long should the throttle period be?

Deciding on the throttle period requires some analysis of user needs. As the previous blog reader example demonstrates, there is a range of periods that might be required:

- For background synchronization, the period can be a few minutes if resource constraints apply, although it should ideally be in the order of 10 seconds. If it is several minutes, users should be kept informed with appropriate *Progress Indicators* (Chapter 14), lest they quit and lose work before the upload kicks in. Ideally, there should be an *Explicit Submission* (later in this chapter) mechanism as well, to allow immediate saving.
- At the other end of the spectrum, for low-level interaction while the user types and mouses around, the period must be in the order of 100 milliseconds.

Real-World Examples

Google Suggest

Google Suggest (<http://www.google.com/webhp?complete=1>) features *Suggestions*, so when you type “A,” the browser pops up a list of popular searches beginning with “A.” To prevent against excessive queries, Google Suggest (<http://serversideguy.blogspot.com/2004/12/google-suggest-dissected.html>) uses Submission Throttling.

Zuggest

Zuggest (<http://www.francishanahan.com/zuggest.aspx>) is a *Live Search* (Chapter 14) showing Amazon results as you type (Figure 10-5). So type “Ab” and you’ll get results like “Basic Ab Workout for Dummies” and “Absolutely Fabulous.” The results are images as well as text, so it would be expensive to search for something you weren’t interested in. So, if you’re searching for “Absolutely,” it’s best to avoid searching for “Ab” and “Abso” and “Absolut” along the way, which is the kind of thing the Google Suggest algorithm would do.

To ensure searches are relevant, Zuggest applies a delay while typing. The assumption is that you’ll be typing at a rate of at least one character per second. Any time you hit a key, you’ll see a “Waiting until you’re done ...” message, and you’ll have a second to hit another key. If no key is pressed, the application assumes you were looking for the current term, and performs a remote call.

As explained in the *Fat Client* pattern, the wiki demo (<http://ajaxify.com/run/wiki>) throttles in a similar manner.

Gmail

Gmail (<http://gmail.com/>) has an auto-save feature that periodically uploads a message being composed.

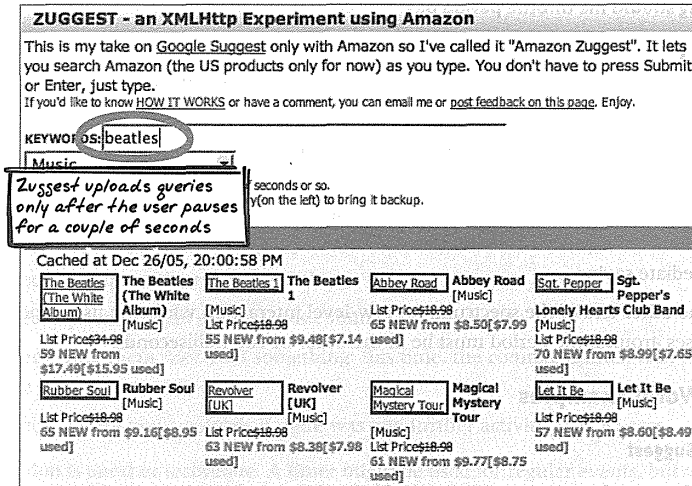


FIGURE 10-5. Zuggest

Prototype framework

Prototype (<http://prototype.conio.net/>) offers a reusable component, `TimedObserver`, which performs Submission Throttling. `ListSomething` (<http://listsomething.com/>), a search engine for classified ads, utilizes `TimedObserver` for its *Live Search*.

Code Example: AjaxPatterns Assistive Search

The Assistive Search demo (<http://ajaxify.com/run/assistiveSearch>) throttles in a similar manner to Google Suggest and other Ajax search apps.

`requestValidCategoriesLoop` runs repeatedly; the precise interval (in milliseconds) is determined by the `THROTTLE_PERIOD` constant (about 100 milliseconds). The last server query is always stored, and there's nothing to do if the current query remains the same as the previous query. If there has been a change, the new query is submitted to the server:

```
function requestValidCategoriesLoop() {
  if (query() != latestServerQuery) {
    vars = {
      queryType: "getValidCategories",
      queryText: escape(query())
    }
    ajaxCaller.get("categories.php", vars, onValidCategoriesResponse,
      false, null);
    latestServerQuery = query();
  }
  setTimeout('requestValidCategoriesLoop();', THROTTLE_PERIOD);
}
```

Related Patterns

Periodic Refresh

Periodic Refresh (see earlier) is somewhat the reverse of Submission Throttling: instead of periodically uploading user input, Periodic Refresh periodically downloads server state.

Progress Indicator

A common trend while performing a periodic update is to include a small *Progress Indicator* (Chapter 14), often as a *Popup* with a message such as a “Saving.”

Metaphor

Science-fiction writers have speculated that interstellar communication would work like this. Because the trip is so great, each imperial command must contain a big load of information—the inter-galactic emperor can’t just tell the colonials to “go fetch a kettle” ... (20 light-years later) ... “now add some water”

Explicit Submission



Submit, Packet, Performance

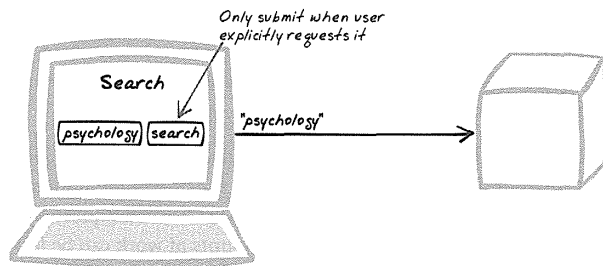


FIGURE 10-6. Explicit Submission

Developer Story

Devi’s producing an Ajax chat web site. She decides text will only be transmitted after the user explicitly clicks a “Done” button.

Problem

How can information be submitted to the server?

Forces

- It's difficult for the server to cope with lots of messages at once.
- Each message has overheads, such as packet headers, and requires some processing at each stage of the browser-server round-trip.
- Users often need to manipulate a small work unit privately, and then upload it as a whole atomic unit to the server.

Solution

Instead of automatically submitting upon each browser event, require the user to explicitly request it; e.g., submit upon a button click. Typically, the user performs some work for a few seconds or minutes, and then clicks a button to tell the server. It's a familiar pattern on the Web as it feels similar to standard form submission.

The most common example is a text field, which may be an input or textarea control. While it would be possible to transmit keystrokes as they happen, that's often not desirable. In a wiki, for instance, imagine what would happen if a user deleted a paragraph in order to replace it. Any other user viewing the wiki would see the paragraph disappear! Furthermore, the history would reflect that transition state.

Where there's only one input field, it sometimes makes sense to rely on `onchange` or `onblur` events to detect that the change has been made. But how about when there are several closely related fields? Even then, some automatic submission is okay, to provide some validation information, for instance. However, if the information is important enough to cause a change to the server, Explicit Submission is a good way to ensure the user intended what's been uploaded.

As well as some inconvenience, the downside of relying on the user to explicitly submit data is...what if the user doesn't? Have you ever quit the browser and forgot to submit a form you were working on? The consequences may be minor for a query-and-report application, but for data entry applications, large amounts of work can be lost when the user forgets to submit it or doesn't realize it's necessary to do so. For that reason, Explicit Submission can sometimes be complemented by automated *Submission Throttling*. For instance, Gmail (<http://gmail.com>) will only send mail upon an Explicit Send command, but nonetheless has an Autosave feature that will periodically upload an in-progress message as a draft.

Decisions

How will the user request the submission?

The submission mechanism should ideally be similar to those in similar non-Ajax systems.

Buttons are one common idiom, with a generic label like Submit, Done, or Go!. More meaningful names are usually clearer, being specific to the task being conducted—for instance, "Buy," "Search," or "Delete." Buttons have the benefit of making the Explicit Submission

mechanism stupidly obvious: “I click the button, the info’s submitted; I don’t click it, it’s not submitted.”

Relying on a significant keystroke, usually Enter, can also work in the right context, and, by using the keyboard, it supports power users.

Listening for the `onblur` is another explicit technique, albeit with a submission mechanism that is not apparent from the UI. The nice thing about `onblur` is that the application can continue submitting without interrupting the usual flow of events. On a *Live Form*, for instance, users can force a submission with Tab or Shift-Tab, which they would have hit anyway to move out of the tab. It also means the user can click somewhere else on the mouse to force a submission. The downside is the risk of accidental submission. There’s also the question of what the user can do to prevent a submission occurring, having already begun typing something? Perhaps they can blank the field, but clearly, communicating this to the user is not easy.

How will you deal with frequent submissions?

Explicit Submission leaves open the possibility of too many submissions at once. This might happen because the user is working too fast, or simply because he has grown impatient and begun banging on the mouse button or the Enter key. A few coping strategies are as follows:

- Limit frequency of submissions with *Submission Throttling* (earlier in this chapter). While that pattern emphasizes automated submissions, it’s still possible to throttle Explicit Submissions.
- After a user has made a submission, use *Page Rearrangement* (Chapter 5) to remove the submission button.
- Soothe impatient users with *Progress Indicators* (Chapter 14) and *Guesstimates* (Chapter 13).
- Prevent excessive delays with Performance Optimizations like *Predictive Fetch* (Chapter 13).
- Show the initial Submission control in a *Popup* (Chapter 15), and then close it upon submission. This is sometimes used for login forms.

No matter how you prevent multiple submissions, be sure to design the server so it can cope with them. *RESTful Service* (Chapter 9) helps here because it can handle such situations gracefully; e.g., by rejecting any invalid queries.

Real-World Examples

Lace Chat

Brett Stimmerman’s Lace Chat (<http://www.socket7.net/lace/>) is an Ajax chat application. Lace users type the entire message and then explicitly submit with a Say button. You can also hit Enter after typing the message. Most chat applications work this way. It’s a little more efficient, but the main benefit is actually for the user. It might be confusing to other

users to see a partially constructed message, so the composer of that message should rectify any errors before posting the message.

The Fonz

“The Fonz” text adventure (<http://www.mrspeaker.webeisteddfod.com/2005/04/17/the-fonz-and-ajax/>) is a command-line game (Figure 10-7). In typical command-line fashion, you type something and submit the whole command at once. Interestingly, command lines don’t have to work that way—it’s feasible to provide hints or some validation support as the user types. The Assistive Search Demo (<http://ajaxify.com/run/assistiveSearch/>) illustrates this point.

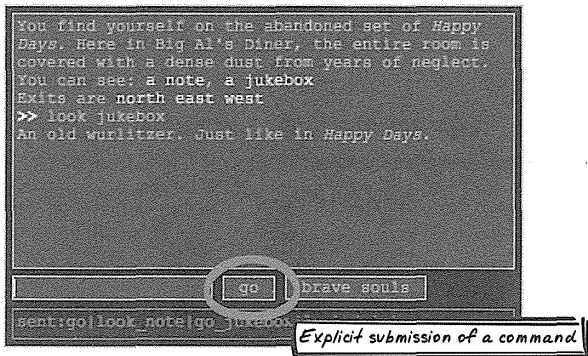


FIGURE 10-7. The Fonz

A9

A9 (<http://a9.com>), as with many search engines, requires Explicit Submission before it searches for the user’s query (Figure 10-8).

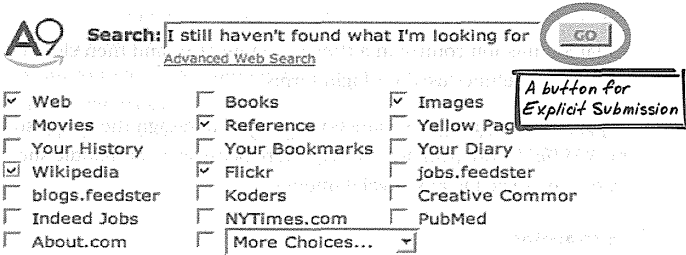


FIGURE 10-8. A9 search

Code Refactoring: AjaxPatterns Form-Based Sum

The Basic Sum Demo (<http://ajaxify.com/run/sum>) illustrates Explicit Submission. In the basic case, there are some input fields and a button, just sitting in a div:


```

<div>

  <div id="figure">
    <input type="text" class="figure" id="figure1"
      name="figure1" size="3" value="4"/><br/>
    <input type="text" class="figure" id="figure2"
      name="figure2" size="3" value="6"/><br/>
    <input type="text" class="figure" id="figure3"
      name="figure3" size="3" value="" /><br/>
  </div>

  <input type="button" id="addButton" value="Add" />

</div>

```

The button is configured to submit the form when it is clicked, an example of Explicit Submission:

```

$("addButton").onclick = function() {
  submitSum();
}

```

We now refactor to an alternative form of Explicit Submission, in which a standard form (<http://ajaxify.com/run/sum/form>) makes the submission. It's still an "Ajax submission" involving no page refresh, but it leverages the standard form mechanism. There are two reasons you might do this in real life: it's a step towards graceful degradation, since non-JavaScript browsers will require the data to be held in a standard form; it will "feel" more like a form to the user—e.g., the Enter key will automatically submit, and any CSS styling for forms will apply to it.

The initial HTML has been wrapped by a `form` tag. With standard JavaScript enabled, the action URL makes no difference because it will never be accessed, but if we want, we could point it to a conventional server-side script that would process the form in the event that JavaScript is turned off. The regular button control has been replaced by a submit button:

```

<form action="http://ajaxify.com/run/sum/form/">

  <div id="figure">
    <input type="text" class="figure" id="figure1"
      name="figure1" size="3" value="4"/><br/>
    <input type="text" class="figure" id="figure2"
      name="figure2" size="3" value="6"/><br/>
    <input type="text" class="figure" id="figure3"
      name="figure3" size="3" value="" /><br/>
  </div>

  <input type="submit" id="addButton" value="Add" />

</form>

```

The script hooks into `onsubmit`, which will be called when the new submit button is clicked. It arranges for an XMLHttpRequest submission via `submitSum()`, then returns false to prevent a conventional form submission and page refresh.

```

$("sumForm").onsubmit = function() {
    submitSum();
    return false;
}

```

Alternatives

Submission Throttling

Submission Throttling (see earlier) talks about automated, periodic submissions, where the present pattern is about the user forcing submissions to take place. The patterns can certainly be combined. Consider text editors that often harden their explicit Save mechanisms with automated backup. An Ajax App can use automated submissions, but allow for manual intervention when the submission must take place NOW.

Related Patterns

Live Form

Live Forms (Chapter 14) often use `onblur` and `onfocus` events, a subtle type of Explicit Submission.

Progress Indicator

Follow up submissions with a *Progress Indicator* (Chapter 14) to give some feedback.

Metaphor

An author drafts each chapter of a book in entirety before submitting it for editorial review.

Distributed Events



Events, Feed, Listener, Messaging, MVC, Notification, Observer, PublishSubscribe, RSS, Refresh, Semantic, Synchronise, Synchronize, Update, WebFeed

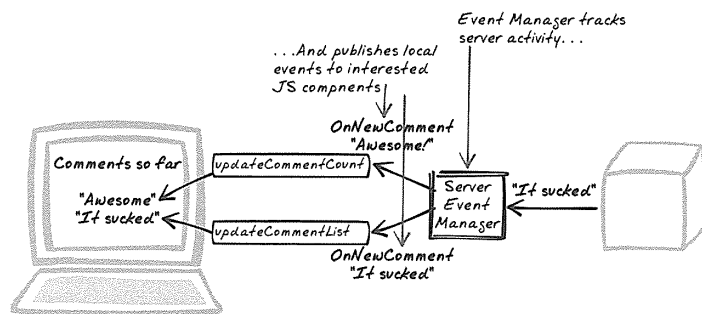


FIGURE 10-9. Distributed Events

Developer Story

Devi's producing a web app for auditors at a financial institution, aiming to highlight substantial transactions. Relevant transactions are already published on the enterprise messaging system, and Devi transforms it into an RSS feed on the web server. The browser script then checks the feed every few seconds, and updates the view whenever it detects a substantial transaction occurred.

Problem

How do you decouple code in a complex application?

Forces

- Ajax Apps involve at least two tiers: a browser tier and a web server tier. In practice, the web server tier is often dependent on further tiers and external systems.
- Each tier can be quite complicated, containing many stateful entities (objects, HTML controls, or regular variables).
- The state of all these entities must often be synchronized, in order to keep users and external systems up-to-date. The synchronization needs to occur within a tier as well as across tiers—for example, an HTML table needs to change whenever browser-side user preferences change, but also whenever the server-side database changes.
- Keeping all these objects synchronized can become complex—there are about n^2 possible message paths from one object to another.

Solution

Keep objects synchronized with an event mechanism. This is a classic software pattern applied to Ajax, related to the Observer (Gamma et al., 1995) and “Publisher-Subscribe” (Buschmann et al., 1995) patterns, and also a key feature in the classic “Model-View-Controller” architecture. The DOM already provides an mechanism for low-level events, but the events discussed here are more semantic in nature; i.e., related to business and application concepts such as “account deleted.” Note that the term “event” is used in a broad manner to mean any subscription-based approach that alleviates the need for direct calls from source to destination. Any publish-subscribe messaging mechanism falls under this definition.

Here's a scenario to motivate the concept of events. Say you have 10 objects with interdependent states. That is, when one object changes, any number of the other nine must change accordingly. The naive implementation would endow each object with an understanding of the other nine objects. Each time it's changed, an object would then tell each other object how to update. Each object now knows the other nine intimately. When one changes, the other nine must be updated—a major blow to encapsulation.

As with many programming problems, you can create a better solution by adding another layer of indirection—in this case, an event mechanism. Let each object broadcast changes instead of directly telling others how to respond. The changes should generally occur in

semantic terms—rather than saying “someone’s clicked me,” an object should say “counting module has begin,” or, “transaction has successfully completed.” And let any object register to be notified whenever a message like this occurs. For larger systems, thinking in terms of events is easier as it breaks down the synchronization logic. You have one simple task to make objects broadcast events whenever they occur. And you have a separate task to decide how objects should actually respond to events, if they care about them at all.

On the Web, this pattern can be applied in different ways:

Server-to-browser

Page elements can be kept in sync with server objects.

Browser-to-browser

Page elements can be kept in sync with each other.

Browser-to-server

Server objects can be kept in sync with each other.

Server-to-server

Server objects can be kept in sync with each other.

Browser-to-server and server-to-server are both feasible, but beyond the scope of the Ajax Patterns because they are more about server-side architecture.

As we saw in Chapter 7, JavaScript offers some event-handling support, but there’s no generic event API. To use distributed events then, you’ll need to roll your own event system, which is easier than it sounds. Server-to-Browser propagation works something like this. There’s a Server Event Manager, with two responsibilities. First, it runs a *Periodic Refresh* (see earlier in this chapter) or participates in *HTTP Streaming* (Chapter 6) to check if any server activity has occurred. Second, it offers an event propagation facility so that interested browser entities can register to discover any changes. When a server change occurs, the Server Event Manager constructs an event object based on the server’s output and passes it any interested listener. To do this, the manager must retain a collection of listeners and the event types they are listening to. The minimal set of services would then be:

- `addListener (eventType, listener)`
- `removeListener (eventType, listener)`

These listeners can be callback functions, like the callback function used by `XMLHttpRequest`. Or, they can be objects that contain a function with a standard callback name for the event type being registered, such as `onUpdate()`.

With Browser-to-Browser propagation, you can also have a Central Server Manager to accept events and notifications. Alternatively, each object can be responsible for creating and propagating events specific to itself. That is, each object capable of generating events needs to allow other objects to register themselves for the events.

Observer is a special case of this pattern that arises frequently. The events are not user actions but change notifications. Event listeners are *observing* an object and responding to

its state. Often, it's used to keep state in sync. An HTML table, for example, can render the latest state of a timetable object on the server side.

Finally, note that this pattern is somewhat speculative and open-ended, but the main purpose should be clear: to add a layer of intermediation so that objects can encapsulate their own responses to system activity, rather than being told what to do by other objects.

Decisions about Distributed Events

Will you publish a history or just the current state?

Given the nature of HTTP, server data must be retrieved periodically—the server can't directly call the browser when something happens. If the server exposes only the current state, there are two problems:

Change detection

The browser can only infer a change has occurred by comparing the previous value, or perhaps by using a version ID or timestamp.

Intermediate states

The browser will miss values that occurred between updates.

Neither of these are showstoppers, so exposing only the current state may well be feasible. The alternative is to publish a history of states along with timestamps. That's more work to output and parse, and it also requires some form of storage on the server. If you are following that option, it's worth formatting the changes using a standard feed-based approach such as RSS or Atom. You'll benefit from the abundance of libraries for both browser and server. As a bonus, the service will be generic enough to be used by external clients, if that's a requirement.

For observer-style events, will you propagate the details of what's changed, or just point to the object that's changed?

Often, events concern an object that's changed state. Sometimes, you only need to propagate a pointer to the object; the recipient will then interrogate the object as required. Other times, you need to send the change itself. The latter approach is useful for functionality that requires not just the object's current state, but the nature of the change. For example, imagine you have an auditing function that logs whenever a budget balance has been increased. It will be much easier if the change event indicates that an increase has occurred. Otherwise, it will have to manually compare the budget against its previous state. For server-to-browser events, indicating the change may be better for performance since it may alleviate a follow-up call to the server.

What information will accompany the event notification?

When an event occurs, there's usually several pieces of information to pass to listeners—for example, a source object (an object that's just changed), the nature of the change or

action that's occurred, and meta-information such as event time and unique ID. You can pass this information in different ways:

String message

Pass the information as a single string.

Single event object

Pass in a single event object containing attributes for each piece of information.

Parameter list

Pass the information as separate parameters.

Each style has its strengths. A string message is the most flexible approach and has the benefit of being a portable format that will work on the server as well. Unfortunately, a string message must often be parsed and formatted to convert to and from useful JavaScript values. A single event object is easier to manipulate and, like a string message, can usually be extended without breaking existing code—the callback function still takes a single value. You can create a factory function to create the event and expose properties, so its interface can be made explicit if you so desire. Finally, a parameter list makes for a cleaner callback function implementation, since you don't have to extract variables from a wrapper object. However, a long list of parameters is cumbersome and difficult to maintain.

Will events be processed synchronously or asynchronously?

The simplest way to handle events is synchronously. As soon as something happens, the event manager immediately notifies all interested parties. All this happens in the same thread of execution, so each event handler becomes a bottleneck—the main program flow that triggered the event won't be able to proceed until each event handler has executed.

If some event handlers are slow, you can get more stable performance by handling events asynchronously. Here, the manager maintains a collection of pending events. Each time a new event arises, it simply adds it to the collection and returns—a very quick operation. Using a repeating timer (see *Scheduling* [Chapter 7]), the manager periodically pulls off pending events and notifies listeners.

There are various decisions involved in asynchronous event handling. First, what sort of collection do you use? A queue is most common, to ensure that events are handled in the order they arise. But sometimes a stack is more appropriate, so that if the manager falls behind, at least the most recent events will have been handled. Another decision is scheduling of the event handler, the object that picks events off the collection. The simplest style is a pure repeating timer, but if the handling takes too long (longer than the timer interval), you'll end up with multiple processes picking off events. One way to prevent this situation is to have the event handler monitor its own progress and cease activity after a certain time has elapsed.

Real-World Examples

ActiveMQ Servlet Adaptor

ActiveMQ (<http://activemq.codehaus.org/>) is an open source implementation of Java Messaging Service (JMS) (<http://java.sun.com/products/jms/>), an official Sun standard for enterprise messaging. As such, it provides a way to pass Java objects and strings between different processes, and includes a publish-subscribe mechanism allowing a process to subscribe for all messages in a particular “topic.”

Normally, the processes run server side, but using a servlet adaptor, ActiveMQ effectively gives the web app, through JavaScript, full ability to send and receive messages.

MapBuilder

MapBuilder (<http://mapbuilder.sourceforge.net>) is a framework for mapping web sites, heavily influenced by MVC. The model holds application data such as current maps, positions, and dimensions. The configuration process wires each model to a number of interested widgets, all of which receive notifications when the model has changed.

Dojo Events Library

Dojo (<http://dojotoolkit.org/download>) is a comprehensive framework aiming to simplify JavaScript development. One thing it does is enhance JavaScript’s standard event management. This includes publish-subscribe functionality. You register one or more functions as *publishers* and one or more functions as *subscribers*. When a publisher function is called, all of the subscriber functions will also be called.

LivePage Library

LivePage (<http://twisted.sourceforge.net/TwistedDocs-1.2.0/howto/livepage.html>), mentioned in *HTTP Streaming* (Chapter 6) examples, is a framework based around Distributed Events.

Code Refactoring: AjaxPatterns Distributed Events Wiki Demo

The Basic Wiki Demo (<http://ajaxify.com/run/wiki>) has a single callback function that serves two purposes: to parse the incoming message and to display it to the user. That’s okay for a simple application, but what if we want to scale up the display operation by displaying different messages in different ways or performing some action when a single message has changed? It won’t be a great surprise that Distributed Events are one way to make the browser script more modular, and this refactoring shows how.

Refactoring to an event mechanism

The first refactoring lays the groundwork for a richer message handling by introducing an event mechanism. There are some minor user-interface differences, for coding convenience. For example, instead of a single “synchronize” point, downloading and uploading are split into independent timing mechanisms; there’s no more background color change

while waiting for a message to upload; and a *One-Second Spotlight* (Chapter 16) effect now occurs when a message has updated, to compensate for the loss of color change. Also, note that a different version of `ajaxCaller` is used, which allows a callback object to be specified in addition to a callback function.

A model object has been introduced to track the state of each message and to play the role of an event manager, notifying listeners of changes. One type of listener receives notification of any new messages. The other type receives notification of updates to a specific message. New message listeners are held in a single array. Update listeners are held in an array of arrays, with all subscribers to a particular message held in an array that's keyed on the message ID:

```
newMessageListeners: new Array(),
messageUpdateListenersById: new Array(),

addNewMessageListener: function(listener) {
    this.newMessageListeners.push(listener);
},
addMessageUpdateListener: function(messageId, listener) {
    var listeners = this.messageUpdateListenersById[messageId];
    listeners.push(listener);
},
```

Notification then works by iterating through the collection of relevant listeners:

```
notifyNewMessageListeners: function(newMessage) {
    for (i=0; i<this.newMessageListeners.length; i++) {
        this.newMessageListeners[i](newMessage);
    }
},
notifyMessageUpdateListeners: function(updatedMessage) {
    var listenersToThisMessage =
        this.messageUpdateListenersById[updatedMessage.id];
    for (i=0; i<listenersToThisMessage.length; i++) {
        listenersToThisMessage[i](updatedMessage);
    }
}
```

How do these events arise? The model object must be started manually and will then periodically poll the server:

```
start: function() {
    this.requestMessages();
    setInterval(this.requestMessages, DOWNLOAD_INTERVAL);
}
...
requestMessages: function() {
    ajaxCaller.getXML("content.php?messages", messageModel.onMessagesLoaded);
},
```

As before, the server provides an *XML Message* (Chapter 9) describing all messages. The model steps through each message in the XML, constructing an equivalent JavaScript object. If the message ID is unknown, the new message listeners are notified, and an array of update listeners is also created for this new message. If the message differs from the cur-

rent message with the same ID, it's changed, so all the update listeners are notified. Recall that the message update notification is fine-grained: only listeners to a particular message ID are notified; hence the extraction of a message-specific list of listeners.

```
onMessagesLoaded: function(xml, callingContext) {
    var incomingMessages = xml.getElementsByTagName("message");
    for (var messageCount=0; messageCount<incomingMessages.length;
        messageCount++) {
        var messageNode = incomingMessages[messageCount];
        var content = this.getChildValue(messageNode, "content");
        content = (content==null ? "" : unescape(content));
        var incomingMessage = {
            id: this.getChildValue(messageNode, "id"),
            lastAuthor: this.getChildValue(messageNode, "lastAuthor"),
            ranking: this.getChildValue(messageNode, "ranking"),
            content: content
        };
        var currentMessage = this.messagesById[incomingMessage.id];
        if (!currentMessage) {
            this.messageUpdateListenersById[incomingMessage.id]=new Array();
            this.notifyNewMessageListeners(incomingMessage);
        } else if (!this.messagesEqual(incomingMessage, currentMessage)) {
            this.notifyMessageUpdateListeners(incomingMessage);
        }
        this.messagesById[incomingMessage.id] = incomingMessage;
    },
    getChildValue: function(parentNode, childName) {
        var childNode = parentNode.getElementsByTagName(childName)[0];
        return childNode.firstChild == null ? null : childNode.firstChild.nodeValue;
    },
    messagesEqual: function(message1, message2) {
        return message1.lastAuthor == message2.lastAuthor
            && message1.ranking == message2.ranking
            && message1.content == message2.content;
    }
}
```

A new `messagesDiv` object has also been created to encapsulate the message-handling logic. On startup, it subscribes for notification of new messages. For each message, it performs a similar function to what was previously done on each update: it creates all the message information and appends to the page, along with a newly introduced visual effect (courtesy of Scriptaculous; see <http://script.aculo.us>).

```
start: function() {
    messageModel.addNewMessageListener(this.onNewMessage);
},
...
onNewMessage: function(message) {
    var messageArea = document.createElement("textarea");
    messageArea.className = "messageArea";
    messageArea.id = message.id;
    messageArea.serverMessage = message;
    ...
    messageDiv.appendChild(lastAuthor);
    messageDiv.appendChild(messageArea);
}
```

```

    ...
    $("messages").appendChild(messageDiv);
    Effect.Appear(messageDiv);
    ...
}

```

The `messageDiv` has another responsibility: it must update the display when a message has updated. Thus, it registers itself as a listener on each message. The easiest way to do this is upon adding each new message:

```

onNewMessage: function(message) {
    ...
    messageModel.addMessageUpdateListener(message.id, function(message) {
        var messageDiv = $("messageDiv" + message.id);
        var lastAuthor = messageDiv.childNodes[0];
        var messageArea = messageDiv.childNodes[1];
        if (messageArea.hasFocus) {
            return;
        }
        lastAuthor.innerHTML = message.id + ". "
            + "<em>" + message.lastAuthor + "</em>" + ". "
            + message.ranking;
        messageArea.value = message.content;
        Effect.Appear(messageDiv);
    });
},

```

Compared to the previous version, we're now only redrawing a message when it's actually changed. Using an event mechanism has helped to separate the logic out. Now, it's the `messageDiv` itself that decides how it will look after a message comes in, which is much more sane than the message-receiving callback making that decision.

Introducing a watchlist

The refactoring above wouldn't be very useful if we stopped with an event mechanism. Good for your work experience perhaps, but we haven't yet added any functionality to justify the effort; it's basically the same application as before. Not to worry! Watchlist Wiki Demo (<http://ajaxify.com/run/wiki/events/watchlist>) to the rescue! A new watchlist monitors interesting messages, so that when a message you're watching is updated (by you or someone else), the watchlist will add a summary line.

To start with, the HTML now includes a watchlist table:

```

<div id="summary">
  <table id="watchlist">
    <tr>
      <th>Author</th>
      <th>Message</th>
    </tr>
    <tbody id="watchlistBody"></tbody>
  </table>
</div>

```

Which messages are in your watchlist? That's determined by a new checkbox control, one per message:

```
onNewMessage: function(message) {
  ...
  var watching = document.createElement("input");
  watching.type = "checkbox";
  watching.messageId = message.id;
  watching.onclick = onWatchingToggled;
  ...
}
```

When the user wants to watch a message, she selects its checkbox. A single function updates the watchlist for all chosen messages. Remember that message update events are fine-grained, so we need to ensure this callback is registered to receive notifications for all the chosen messages and nothing else. So when a user deselects a message, we'll unregister the function as a listener on that message. Note that this functionality necessitated the creation of an function to unregister listeners, which was never required in the previous version.

```
function onWatchingToggled(event) {
  event = event || window.event;
  var checkbox = event.target || event.srcElement;
  if (checkbox.checked) {
    messageModel.addMessageUpdateListener(checkbox.messageId, onWatchedMessageUpdate);
  } else {
    messageModel.removeMessageUpdateListener(checkbox.messageId,
      onWatchedMessageUpdate);
  }
}
```

onWatchedMessageUpdate will now receive notification of any new messages that are being watched. It simply adds a summary row to the table and runs a visual effect:

```
function onWatchedMessageUpdate(message) {

  var summary = message.content;
  if (summary.length > 35) {
    summary = summary.substring(0, 15) + "..."
      + summary.substring(summary.length - 15);
  }

  var row = document.createElement("tr");

  var authorCol = document.createElement("td");
  authorCol.className = "authorSummary";
  authorCol.innerHTML = message.author;
  row.appendChild(authorCol);

  var contentCol = document.createElement("td");
  contentCol.className = "contentSummary";
```

```

        contentCol.innerHTML = summary;
        row.appendChild(contentCol);

        if ($("#watchlistBody").childNodes.length > 0) {
            $("#watchlistBody").insertBefore(row, $("#watchlistBody").childNodes[0]);
        } else {
            $("#watchlistBody").appendChild(row);
        }
        Effect.Appear(row);
    }
}

```

We now have two independent functions that receive notifications of new messages arriving from the server. Each can use the information however it pleases. This is a much more scalable approach than having the server message recipient dictate how the browser should respond.

Related Patterns

Periodic Refresh, HTTP Streaming

For server-to-browser propagation, *Periodic Refresh* (see earlier) or *HTTP Streaming* (Chapter 6) is required.

RESTful Service

Distributed Events usually involve a browser element observing a server-side entity. REST is ideal for this purpose as it provides a simple, standard way to exposes server state.

XML Data Island

If the server responds with XML and you need to retain state locally—e.g., to track differences—an *XML Data Island* (Chapter 11) would be useful. Under some technologies illustrated in that pattern, XML Data Islands allow for automated updates—when the data island changes, then a control is updated, and vice versa.

Metaphor

The old newspaper analogy still works. People can subscribe to any number of newspapers, and each newspaper can have any number of subscribers. The algorithm does not explicitly mention any particular subscriber; rather, when a newspaper comes out, it simply loops through each subscriber and sends a copy to each of them.

Cross-Domain Proxy



Aggregator, Fusion, Mash-Up, Mashup, Mediator, Mix, Proxy, Tunnel

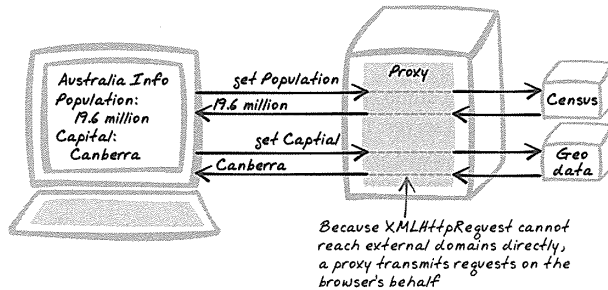


FIGURE 10-10. Cross-Domain Proxy

Developer Story

Dave's working on a job-hunting web site and wants to "mash up" content from various other sites. Each job ad is accompanied by a review from a corporate forum web site, recent company news, and a stock ticker. To get these details into the browser, the server queries several sites and exposes their content as a web service that the browser can query.

Problem

How can you augment your application with services available elsewhere on the Web?

Forces

- There's a lot of data and useful services on the Web. Much of it is freely available. It's usually more effective to leverage that content than to replicate it yourself.
- The "same-origin policy" constrains most Web Remoting (Chapter 6) techniques, meaning that the browser script can only talk to the server from whence it came, and not to anyone else.
- External servers often use protocols and message formats that would be difficult for browser-side scripts to process. They may also require authentication, and such details should not be exposed in the browser.
- To comply with service agreements, web site owners often have to control and monitor all traffic to and from their web service, which is impossible if their server is being bypassed.

Solution

Create proxying and mediating web services to facilitate communication between the browser and external domains.

As explained in *XMLHttpRequest Call* (Chapter 6), the same-origin policy means the browser script can only talk to the server from whence it came, which is the “base server.” Hence, any communication with external domains must go via the base server.

The simplest form of proxy is a dumb *Web Service* (Chapter 6) that simply routes traffic between browser and external server. The service can accept the remote URL as a parameter. That URL will then be accessed synchronously, and the service will output its response. All this follows the Proxy pattern (Gamma et al., 1995).

A cleaner type of Cross-Domain Proxy is more closely based on the Facade or Adaptor pattern (Gamma et al.). In this case, the base server presents the interface that’s most suitable to the browser script, and performs whatever manipulation is necessary to interact with the external script. For example, the external server might present a complex, over-engineered, SOAP-based *RPC Service*, but you want the browser to deal in simple *Plain-Text Messages* (Chapter 9). The likely solution is to present the plain-text *Web Service*, dress up requests into SOAP messages, and undress responses into Plain-Text Messages.

The Facade/Adaptor approach is generally better for two reasons. First, it keeps the client simple, protecting it from dealing with the details of the protocol used to communicate with the external server. Second, it’s more secure: the first approach will allow any client to call out to any server they feel like, whereas the second approach will allow you to exert much more control over what kind of communication takes place.

A Cross-Domain Proxy is implemented using some form of HTTP client library to access the remote server. Generally, the connection should be quite straightforward—specify the URL and grab the response as a string. However, it might get more complex if the remote web service relies on cookies and authentication, parameters that the proxy might have to pass on from the Ajax request that initiated the call.

Decisions

What external content will be accessed?

In theory, anything you can do with a browser can theoretically be accomplished by an automated HTTP client. However, there are likely to be legal constraints, and technical challenges too if there is serious usage of JavaScript or browser plugins. Furthermore, your script will be vulnerable to changes in the site layout, and many content providers have deliberately performed subtle changes to prevent automated access. Thus, relying on the ever-increasing collection of public APIs and structured data would make a better choice.

You can find a collection of public APIs at <http://wsfinder.com>. Some popular APIs include Amazon, Delicious, EvDB, Flickr, Google Maps, Google Search, Technorati, and Yahoo Maps.

How will you connect to the external server?

If you're scraping content directly from a public web site, you'll need to use the underlying web protocol, HTTP. And even if you're talking to an API, you'll probably be communicating with HTTP anyway. In some cases, API publishers will provide code to access the content, rather than just publishing the specification. Also, with more complex web services protocols like SOAP, you don't have to write a lot of low-level code yourself. In many cases, though, the easiest thing to do is talk HTTP and manually format and parse messages yourself.

Many scripting languages feature built-in libraries to communicate in HTTP and are often well-equipped for Cross-Domain Proxy implementations due to strong support for regular expression manipulation. A PHP example is featured in the code example that follows shortly.

In the Java world, for instance, the standard API provides some relatively low-level support for HTTP clients, but you can use some of the web site testing libraries to quickly extract content from external sites. HttpUnit is a good example, and it also has some support for JavaScript manipulation. To grab some content:

```
import com.meterware.httpunit.WebConversation;
import com.meterware.httpunit.GetMethodWebRequest;
import com.meterware.httpunit.WebRequest;

public class PatternSorter {

    public String getContent(String url) {
        try {
            WebConversation wc = new WebConversation();
            WebRequest request = new GetMethodWebRequest(url);
            String response = wc.getResponse(request).getText();
        } catch (Exception ex) {
            throw new RuntimeException("Could not get content from " + url, ex);
        }
    }

    public static void main(String[] args) {
        System.out.println("http://ajaxpatterns.org");
    }
}
```

How will you deal with errors and delays in accessing the service?

At times, there will be errors accessing an external web service. Think twice before attributing blame because the problem might be at your end, or somewhere along the network. This has implications for any error messages you might show users. Ideally, you should be

able to detect that an error has occurred, and then log the details for immediate attention. In responding to the user, you have several options:

- Ensure you have a good server-side framework in place to detect errors, so they're not directly—ignorantly—passed on to users. In addition, be sure to detect timeouts and handle them proactively; e.g., respond with an appropriate error message.
- In some cases, you will simply have to admit failure and show an error message to users. If possible, suggest an alternative or tell them when they should try again.
- Provide reduced functionality.
- Switch to a prearranged substitute web service.
- Rely on cached results, if they're still relevant.

Under what licensing terms will you access the remote service?

The legals of invoking *Web Services* (Chapter 6) are tricky, vaguely-defined, and always evolving. Some services are open for public use, others require you hold an API key, and others are critical enough to warrant authentication via digital signatures. In each case, there are usage terms involved, even if they're not explicit, and you will need to investigate issues such as the authentication mechanism, the number of queries per day, how the data will be used, server uptime, and support level.

Real-World Examples

WPLicense

The WPLicense Wordpress plugin (<http://yergler.net/projects/wplicense>) presents the blogger with some forms to specify their preferred license statement, based on the Creative Commons model. The server is acting as a middleman between the browser and the Creative Commons API (<http://api.creativecommons.org/rest/1.0/>).

Housing Maps (Craigslist and Google Maps)

Housing Maps (<http://housingmaps.com>) is a mashup between Google Maps (<http://maps.google.com>) and Craigslist (<http://craigslist.com>), a community-oriented classifieds advertising web site. What does that mean? It means you get to see a list of advertised homes on one side and view a map of those homes on the other side, using the familiar Google Maps thumbtacks to pinpoint the locations of the advertised homes. (See Figure 10-11.)

A few features:

- Click on a thumbtack and you'll see a *Popup* (Chapter 15) balloon appear, showing a summary of the classified ad: price, terms, address, and photos. The balloon contains a hyperlink to the ad at Craigslist
- Change a category (e.g., price) and you'll see both the thumbtacks and home lists update. Change the city and you'll see the map change as well.
- You can pan and zoom as on Google Maps.

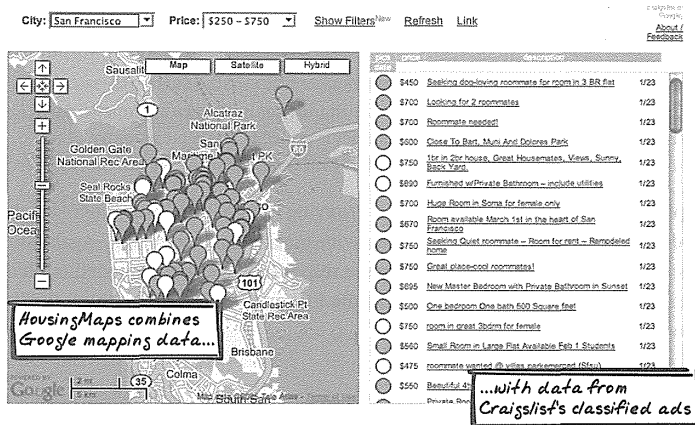


FIGURE 10-11. Housing Maps

Cross-Domain Proxy is used to grab housing data from Craigslist, while the map images are fetched directly from Google's server into the browser.

Bill Gates Wealth Clock

This was not Ajaxian, but noteworthy as the first prominent mashup application and a precursor to the Ajaxian Cross-Domain Proxy pattern. Web guru Philip Greenspun mixed data from several sources to produce a "wealth clock" showing Bill Gates' worth at any time (http://groups-beta.google.com/group/nf.general/browse_thread/thread/bc7fc7bfab19937f/88fa32f2cf6dd6bb). This is no longer working, unfortunately (<http://rhea.redhat.com/bboard-archive/webdb/0006ad.html>), but the code is still available (<http://philip.greenspun.com/seia/examples-basics/wealth-clock.tcl.txt>). See Figure 10-12.

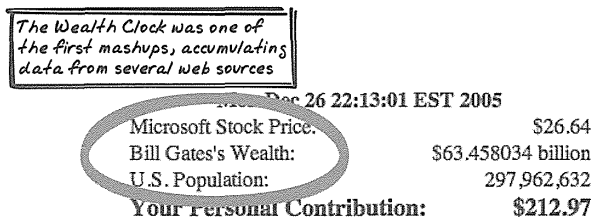


FIGURE 10-12. Bill Gates Wealth Clock

The price per stock was extracted from Yahoo! Finance. The number of stocks in Gates' portfolio came from Microsoft reports (it was not automatically extracted). And the U.S. population came from the U.S. Census Bureau.

Here's what Greenspun said back in 2001 (<http://philip.greenspun.com/teaching/teaching-software-engineering>):

(M)any of the most interesting and challenging problems of the coming decades will center around the Internet and Web as a distributed computing environment. An "island" site such as amazon.com or photo.net is probably less representative of the future than <http://www.webho.com/WealthClock> (Bill Gates Personal Wealth Clock).

Another prominent usage of Cross-Domain Proxying in "Web 1.0" was meta-searching, as performed by crawlers such as MetaCrawler.com (<http://metacrawler.com>). Again, all the work was done in the server, and the browser response was usually opened while the results came back to the server, so at least some information could be shown while requests were still out.

CPaint library

CPaint (<http://cpaint.sourceforge.net/>) is a Web Remoting (Chapter 6) library. A special "proxy" URL can be established (http://cpaint.sourceforge.net/doc/frontend.class.cpaint.set_proxy_url.html), pointing to a proxy on the originating server, so that remoting to external domains has virtually the same API as to the originating server.

Code Example: WPLicense

See the Code Example in *Live Form* (Chapter 14) for a full background on the WPLicense's license selection process, but here's a quick summary:

- A field lets the user select a license type (e.g., Creative Commons versus Public Domain versus Sampling).
- Once the license type is chosen, it's immediately sent to the server, and the server responds with a form consisting of questions relevant to this particular type. If the license is Creative Commons, for example, one of the questions is "Allow commercial uses of this work?"
- Each time the user changes one of the license options, the server updates some hidden fields. These fields will be uploaded to the server, to be persisted when the user eventually submits the form.

What all this glosses over is the Cross-Domain Proxy pattern that goes on behind the scenes. At each of these three stages, the server is actually interacting with CreativeCommons.org via its public API (<http://api.creativecommons.org/rest/1.0/>), which you can read about at <http://api.creativecommons.org>. The plugin provides a clean separation between web server logic and cross-domain mediation: a separate PHP file (`ccwsclient.php`) hosts several API facade operations, accepting and returning data structures in standard PHP style. This client in turn delegates all the XML infrastructure stuff to a third-party library MiniXML, at <http://minixml.psychogenic.com>.

Let's now zoom in on the three steps.

1. Retrieving license types

The server doesn't have the license types hardcoded. Instead, it retrieves them via a web service. This is the core of the Cross-Domain Proxy functionality—the PHP function `fs` will retrieve content from the specified URL:

```
$WS_ROOT = "http://api.creativecommons.org/rest/1.0/";  
...  
$xml = file_get_contents($WS_ROOT);
```

In fact, you can see exactly what's pumped into `$xml` by visiting <http://api.creativecommons.org/rest/1.0/>, and then choose View Page Source. Alternatively, use a command-like application such as `curl` (`curl http://api.creativecommons.org/rest/1.0/`), which yields the following XML (reformatted):

```
<licenses>  
  <license id="standard">Creative Commons</license>  
  <license id="publicdomain">Public Domain</license>  
  <license id="recombo">Sampling</license>  
</licenses>
```

Once we know what license types are available, it's a matter of transforming the XML into an HTML selector. XSLT could be used here, but it's just as easy to do it manually:

```
foreach($license_classes as $key => $l_id) {  
  echo 'option value="' . $key . '" >' . $l_id . '</option>';  
};
```

2. Retrieving license questions

Presenting the questions gets interesting, because the server must now present questions and accept answers without the programmer knowing in advance what those questions will be.

Browser requests for license questions invoke a URL that depends on the user's chosen license type. If the license type is Creative Commons, which has the ID "standard," the URL is <http://api.creativecommons.org/rest/1.0/license/standard/>. Visit that URL, and you'll see the questions and possible answers. For example:

```
<field id="derivatives">  
  <label xml:lang="en">Allows modifications of your work?</label>  
  <description xml:lang="en">The licensor permits others to copy, distribute and  
    perform only  
    unaltered copies of the work, not derivative works based on it.</description>  
  <type>enum</type>  
  <enum id="y">  
    <label xml:lang="en">Yes</label>  
  </enum>  
  <enum id="sa">  
    <label xml:lang="en">ShareAlike</label>  
  </enum>
```

```

<enum id="n">
  <label xml:lang="en">No</label>
</enum>
</field>

```

Equipped with all that information about each question, the server must now transform it into an HTML selector. Ultimately, a loop is used to traverse the entire data structure and generate a selector for each field:

```

foreach ($fields as $f_id=>$f_data) {
  $result .= '<tr><th><nobr>' . $f_data['label'] . '</nobr></th><td>';
  // generate the appropriate widget
  if ($f_data['type'] == 'enum') {
    $result .= '<select class="lic_q" id="' . $f_id . '" lic_q="true" size="1">';
    foreach ($f_data['options'] as $enum_id=>$enum_val) {
      $result .= '<option value="' . $enum_id . '">' . $enum_val . '</option>';
    } // for each option
    ...
  }
}

```

As explained in the *Live Form* (Chapter 14) discussion, this new form HTML is directly placed onto a div.

3. Handling user answers

The Creative Commons' `issueLicense` web service accepts XML input of all the license criteria, and then outputs a document containing a URL for the license criteria, along with some other, related information. In a similar manner to the previous code, all of this information is transformed into HTML. This time, it's used to populate several fields, rather than generate any new input fields.

Alternatives

On-Demand JavaScript

A fairly old cross-domain technique is to use *On-Demand JavaScript*; see the discussion of *Cross-Domain Loading* for that pattern in Chapter 6. The main benefit over Cross-Domain Proxy is reduced resources—the base server is bypassed, so there's no bandwidth or processing costs involved. However, there's a major constraint: the server must expose a suitable script to fetch, because it's not possible to just extract arbitrary information from a server. Additional problems include lack of server-side logging, inability to reach services that require authentication, and the security concerns described in On-Demand JavaScript.

Shared document.domain

When we speak of the “same-origin” policy, we're not necessarily referring to the true domain a document is served from; each document has a mutable `document.domain` property (for example, `document.domain`) that turns out to be the critical factor in cross-domain calls. If two documents declare the same domain property, regardless of their true origin, they should be able to communicate with each other using `XMLHttpRequest`. Jotspot developer Abe Fettig has explained how to exploit this knowledge for making cross-domain communication practical

([http://fettig.net/weblog/2005/11/28/how-to-make-xmlhttprequest-connections-to-another-server-in-your-domain/XMLHttpRequest Call](http://fettig.net/weblog/2005/11/28/how-to-make-xmlhttprequest-connections-to-another-server-in-your-domain/XMLHttpRequest%20Call)). The trick relies on embedding the external document and having that document—as well as your own document—define the same `document.domain` property. Thus, as with the *On-Demand JavaScript* alternative, it does have one key constraint: the external server must explicitly cooperate. In addition, you can't just declare an arbitrary domain; it has to be a "parent" of the true domain (<http://www.mozilla.org/projects/security/components/same-origin.html>; i.e., "shop.example.com" can declare "example.com" but not "anyoldmegacorp.com". Due to these constraints, it's best suited to a situation where there's a direct relationship between the respective site owners.

Images

Images have always worked across domains, with various benefits (syndication) and more than a few problems ("bandwidth theft") along the way. As noted in the "Alternatives" section of *XMLHttpRequest Call* (Chapter 6), images can be used for general-purpose remot-ing, usually with 1-pixel images that will never be rendered. While that's a hack you'll likely not need these days, transfer of legitimate images remains a useful capability. When you run the Google Maps API (<http://www.google.com/apis/maps/>) in your web page, for example, it pulls down map images directly from Google's own servers.

Related Patterns

Performance Optimization patterns

Since external calls can be expensive, the Performance Optimization patterns apply. They may be applied at the level of the browser, or the server, or both. For example, caching can take place in the browser, the server, or both.

Want to Know More?

See WSFinder: a wiki of public web services (<http://www.wsfinder.com/>).

DOM Population

W

HAT DOES THE BROWSER DO ONCE IT RECEIVES A RESPONSE FROM THE SERVER? SEVERAL OPTIONS ARE covered in this chapter. *XML Data Island* explains how you can store an XML document within the web page's DOM instead of converting it into a custom data structure. It also touches on some browser-specific features that let you tie the document's value to the display. XML responses are also the primary concern of *Browser-Side XSLT*, explaining how an XML document can be converted to XHTML for inclusion on the page, or modified for uploading back to the server. *Browser-Side Templating* is not XML-specific; it brings to JavaScript the embedded scripting template concept popular in many server environments, such as JSPs and PHP scripts.

XML Data Island



Data, DOM, Storage, XBL, XML

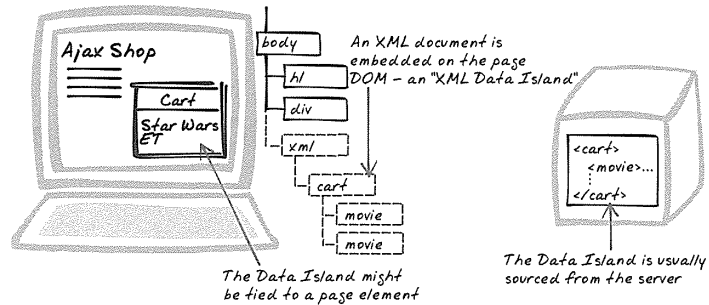


FIGURE 11-1. XML Data Island

Developer Story

Dave's demographics viewer stores each table of statistics as an XML document. A big cache of XML documents is held within the page, one for each table. The user can rearrange the tables and switch them between visible and hidden, but all the XML metadata remains on the page. To render the tables, Dave exploits the browser's ability to translate XML into HTML tables.

Problem

How can you render incoming XML and retain the data?

Forces

- Many Ajax Apps receive *XML Messages* and must convert them into HTML.
- The incoming data often has to be retained as well.

Solution

Retain XML responses as XML Data Islands—nodes within the HTML DOM.

Strictly speaking, an XML Data Island is an XML document embedded in a standard XHTML document, usually within an `<xml>` tag. Say the initial page looks like this:

```
<html>
  <head>
    ...
  </head>
  <body>
    <h1>Top Score</h1>
    <p>Here's the ... </p>
    ...
```



```

        <xml id="scoreData"></xml>
    </body>
</html>

```

After an *XMLHttpRequest* Call (Chapter 6) brings the browser an *XML Message* (Chapter 9), the message is retained in the tag, using techniques described in a moment. The result is a web page DOM like this:

```

<html>
  <head>
    ...
  </head>
  <body>
    <h1>Top Scores</h1>
    <p>Here's the ... </p>
    ...
    <xml id="scoreData">
      <score>
        <title>Galaga</title>
        <player>AAA</player>
        <score>999610</score>
      </score>
    </xml>
  </body>
</html>

```

How do you do something like that in JavaScript? You might assume it's pretty trivial, given that HTML itself is XML (or close enough to it). But unfortunately, it's not as easy as you'd expect and also is browser-dependent. If you have the XML string, you can set `scoreData` element's `innerHTML` as the string value. If you have only a DOM node, you can convert it to an XML string by inspecting its `innerHTML` property. However, this won't work on all browsers and might lead to portability problems as browsers interpret the string in different ways. An alternative is to avoid string manipulation and directly graft one DOM node onto another. The document object has two useful methods here: `cloneNode()` and `importNode()`. Again, there are plenty of portability issues to consider—see Peter-Paul Koch's discussion (http://www.quirksmode.org/blog/archives/2005/12/xmlhttp_notes_c.html). In most cases, the easiest, most portable, solution will be to use a library like Sarissa (<http://sarissa.sourceforge.net>).

In a more general sense, this pattern involves retaining *XML Messages* regardless of how you store them. While the DOM is a convenient place for storage and has some value-added features described below, you could also save the XML in a normal JavaScript variable. The key characteristic of the pattern is that you're retaining the *XML Message* and using it as a data structure, rather than transforming it into a custom JavaScript object.

It's easy enough to do this, but what's the point? Here are three applications:

Transforming XML Data Islands to and from HTML

XML Data Islands are traditionally an IE-specific capability. From IE5, you can tie HTML controls to an embedded XML document. When the XML data changes, the HTML is also updated. And when the HTML changes—if the control is mutable—the

XML is updated. In IE, to tie an HTML element to a data island, you indicate the XML data island, preceded by a # and the field within that island. So to create an input field based on the player field above:

```
<input type="text" datasrc="#scoreData" datafld="player">
```

Now, this is an IE-specific function, but Firefox has its own alternative: eXtensible Binding Language (XBL) (<http://www.zdnetasia.com/techguide/webdev/0,39044903,39236695,00.htm>). The idea is similar, but the updating behavior isn't automatic. With Mozilla, you have to create an XBL document that explicitly states how HTML controls relate to the XML.

Even without these browser-specific capabilities, it's still fairly straightforward to make the transformation using a technology like *Browser-Side XSLT* (see later in this chapter).

Retaining XML for later use

Sometimes it's useful to retain the response for later use. For example, you can keep it to build up a *Browser-Side Cache* (Chapter 13). Another application is for validation—the browser might receive a raw XML document containing some form fields, then augment it with user responses before uploading it back to the server.

To store a response for later use, you could wrap it into a custom data structure. But that requires some coding effort and also complicates things a bit by adding a new data structure. When the response is an *XML Message*, you have an alternative: just keep the XML. While it may not be the optimal format, it's still a structured format that you can query quite easily.

Including an XML document on initial page load

The initial page load sequence sometimes requires an XML document. For example, the XML document might describe the initial data to be viewed or it might be a stylesheet for *Browser-Side XSLT*. You could go through the motions of extracting it with a further *XMLHttpRequest Call*, but it would be faster to load the XML as part of the initial page load. An XML Data Island is an easy way to do this. Just output the XML content wrapped inside an `<xml>` tag. Alternatively, IE lets you specify a URL in the `src` attribute.

Real-World Examples

PerfectXML Demo

Darshan Singh's IE demo (<http://www.perfectxml.com/articles/xml/islandex.htm>) is explained in the corresponding PerfectXML article (<http://www.perfectxml.com/articles/xml/msxml30.asp>). It's a straightforward table, populated by an *XML Data Island* embedded in the corresponding HTML. The demo works in IE only.

Mozilla.org demo

Thad Hoffman's Mozilla demo (<http://www.mozilla.org/xmlextras/xmldataislands/example1.html>) is explained in a corresponding mozilla.org article (<http://www.mozilla.org/xmlextras/xmldataislands/>): corresponding mozilla.org article. It simulates IE's behavior, using standard

XML parsing in JavaScript, to convert XML to HTML. The demo works only in Mozilla or Firefox.

TechRepublic demo

Philip Perkin's example explains how to use Mozilla's answer to XML Data Islands, eXtensible Binding Language (XBL); see Figure 11-2 (http://techrepublic.com.com/5100-3513_11-5810495.html). The code's online and there's a demo available (http://www.phillipweb.com/mozilla/mozilla_xbl.htm).

John	<i>XBL is used to keep input field in sync with the embedded XML's Data Island. You can change either the form or the XML, and the other will update</i>
Doe	
123 Some Street	
Smalltown	
US	
99999	
8005551212	
<pre><XML> <USERINFO> <FIRST_NAME>John</FIRST_NAME> <LAST_NAME>Doe</LAST_NAME> <STREET>123 Some Street</STREET> <CITY>Smalltown</CITY> <STATE>US</STATE> <POSTAL>99999</POSTAL> <TELEPHONE>8005551212</TELEPHONE> </USERINFO> </XML></pre>	

FIGURE 11-2. XBL demo

Code Refactoring: AjaxPatterns XML Data Island Sum

This example adds a very simple caching mechanism to the Basic Sum Demo (<http://ajaxify.com/run/sum>). The most recent XML response is retained as an XML Data Island, so if the user tries to resubmit the query, no server call occurs. In the absence of an XML Data Island, the script would need to retain the call and response in a custom data format. But retaining it as XML means no data format has to be created. Instead of manually navigating the document with the standard DOM API, the Interactive Website Framework (IWF) library (<http://iwf.sourceforge.net>) is used. IWF lets you treat the DOM object more like a custom data structure—as the following code shows, it lets you drill down using tag names rather than generic XML names.

Some minor changes are made to the XML format for easier parsing, leading to the following data format (<http://ajaxify.com/run/sum/xml/dataIsland/sumXML.php?figure1=4&figure2=6>):

```
<sum>
  <inputs>
    <figure1>4</figure1>
    <figure2>6</figure2>
    <figure3></figure3>
  </inputs>
```

```

        <outputs>10</outputs>
    </sum>

```

To the initial HTML, a placeholder has been introduced to retain the most recent XML response—the XML Data Island:

```

<xml id="sumResponse"></xml>

```

The XML is retrieved as a plain-text document and IWF is used to transform it into a special DOM-like format for convenient parsing. The IWF document is interrogated to get the sum, which is injected onto the DOM status element as before. Finally, the XML Data Island is populated with the XML response in its plain-text format (as opposed to a DOM object). The XML response includes the input figures as well as the resulting sum, so we'll be able to use it later on to decide whether the figures have changed since the last response.

```

function onSumResponse(xml, headers, callingContext) {
    var doc = new iwfXmlDoc(xml);
    $("sum").innerHTML = doc.sum.outputs[0].getText()
    $("sumResponse").innerHTML = xml;
}

```

The XML Data Island is used to decide whether a call is necessary. To make the effect more visible, the status area is blanked as soon as the user clicks submit. It's repopulated with the retained value if it turns out the current data is the same as the previous query:

```

function submitSum() {
    $("sum").innerHTML = "---";
    var figures = {
        figure1: $("figure1").value,
        figure2: $("figure2").value,
        figure3: $("figure3").value
    }
    var sumResponseXML = $("sumResponse").innerHTML;
    if (sumResponseXML!="") {
        doc = new iwfXmlDoc(sumResponseXML);
        var alreadyStoredInDOM = (
            figures.figure1 == doc.sum.inputs[0].figure1[0].getText()
            && figures.figure2 == doc.sum.inputs[0].figure2[0].getText()
            && figures.figure3 == doc.sum.inputs[0].figure3[0].getText()
        );
        if (alreadyStoredInDOM) {
            // No need to fetch - just retrieve the sum from the DOM
            $("sum").innerHTML = doc.sum.outputs[0].getText();
            return;
        }
    }
    ajaxCaller.get("sumXML.phtml", figures, onSumResponse, false, null);
}

```

Alternatives

Browser-Side XSLT

Browser-Side XSLT (see the next pattern) is a more general way to transform XML into HTML. In addition, the patterns are related insofar as Browser-Side XSLT can use an XML Data Island to store stylesheets.

Browser-Side Templating

Browser-Side Templating (see later) is a general technique for converting XML or other data formats into HTML.

Browser-Side Cache

XML Data Island can be used to store a *Browser-Side Cache* (Chapter 13).

Metaphor

The name itself is the metaphor: an island of data amid a pool of HTML content.

Browser-Side XSLT



Presentation, Render, Style, Stylesheet, Transform, View, XML, XPath, XSLT

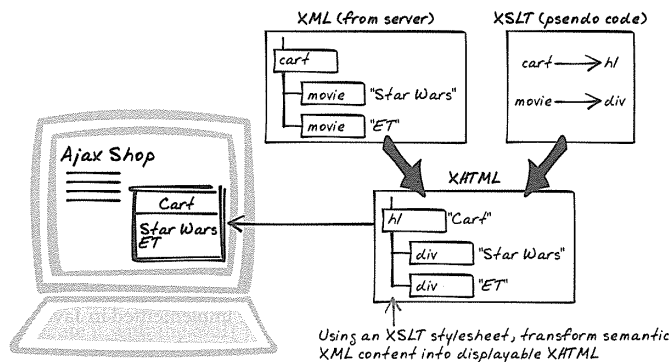


FIGURE 11-3. Browser-Side XSLT

Developer Story

Dave has just received a request to change the weather portlet on the homepage. The image must now be on the bottom, not the top, and the wording should be more precise. The browser receives periodic weather updates in the form of an XML specification, and it uses Browser-Side XSLT to get HTML for the portlet. The stylesheet is embedded inside the static homepage HTML file. So Dave just has to tweak the XSLT to get all the browsers rendering the new design.

Problem

How can you render incoming XML?

Forces

- Many *Web Services* (Chapter 6) output *XML Messages* (Chapter 9), and Ajax clients have to convert them into HTML.
- While modern browsers provide good support for XML parsing via the JavaScript's DOM support, the direct interface is lengthy and tedious to use.
- Code gets complex and error-prone when you mix HTML generation with application logic. JavaScript is not well-suited to programmatically building up HTML strings from input XML.

Solution

Apply XSLT to convert XML Messages into XHTML. XSLT—Extensible Stylesheet Language Transformations (<http://www.w3.org/TR/xslt>)—is a language for transforming XML documents into other forms. Here, XML is transformed to XHTML, a type of HTML that conforms to the XML standard. (You could get away with converting to plain-old HTML, but things work better if you stick to XHTML.) XSLT is fairly well-supported in modern browsers and, although browser APIs vary, there are good cross-browser toolkits available.

XSLT is well-suited to rendering XML. An XML document is designed to encapsulate a data object, and an XSLT stylesheet is a strategy for presenting such objects. Previously, XSLT has been used as a server-side technology—the server grabs some XML from a repository or another process and creates a web page by transforming it into XHTML. All that happens on the server. More recently, browsers have incorporated XSLT functionality, so the browser can automatically create an HTML page by marrying together an XML document with an XSLT stylesheet.

Browser-Side XSLT is slightly different. Here, the XML document does not constitute the contents of the entire page, but it does constitute a response from an *XMLHttpRequest Call* (Chapter 6). The transformation to XHTML can take advantage of the browser's built-in XSLT support, or alternatively an XSLT processor can be implemented in JavaScript, building on the browser's more primitive XML features. However the XSLT occurs, you're unlikely to be implementing it yourself. There are a couple of good cross-browser libraries available, discussed in later in this chapter in "Real-World Examples."

It's beyond the scope of this pattern to explain XSLT in any detail; however, here's a quick overview:

- An *XSLT stylesheet* is itself an XML document.
- An *XSLT processor* transforms an XML document into another form, by parsing the original document and using the stylesheet as the transformation strategy.

- XSLT expressions are specified in another language, XPath. Among its capabilities, XPath provides a powerful mechanism for expressing a position within an XML document. For example, the XPath expression `/category/items/category` will match twice when the following document is processed:

```
<category>
  <items>
    <category name="popular">
      <!-- XPath expression matches this category node -->
      ...
    </category>
    <category name="extras">
      <!-- XPath expression matches this category node -->
      ...
    </category>
  </items>
</category>
```

- A stylesheet is composed of rules. Each rule has a pattern that defines when it applies, and a template that dictates what will be output each time the pattern is encountered. Continuing with the example, here's a full rule:

```
<xsl:template match="/category/items/category">
  <div class="category"
    onclick="retrieveCategory('{@name}')">
    <xsl:value-of select="@name"/>
  </div>
</xsl:template>
```

When each node is reached, the template is outputted. `@name` refers to the `name` attribute on the category tag. So when the processing engine reaches the following XML segment:

```
<category name="extras">
```

the following XHTML will be output:

```
<div class="category"
  onclick="retrieveCategory('{extras}')">
  <xsl:value-of select="extras"/>
</div>
```

This discussion has focused on the most obvious application of Browser-Side XSLT: conversion to HTML for display to the user. You can also convert the XML to JavaScript too, and then execute it with the `eval()` method. For example, it would be possible to build a native JavaScript object by converting some XML into some code that creates the object.

Decisions

How will you obtain the XSLT stylesheet?

Any Browser-Side XSLT processor requires two things:

- An XML document
- An XSLT stylesheet

Both are usually passed in to the XSLT processor as plain strings representing the entire document (as opposed to URLs, say). The XML document usually comes from an *XMLHttpRequest Call* (Chapter 6), but where does the XSLT stylesheet come from? You have a few options.

Store the stylesheet server side

You can hold the file server side, and then use an independent *XMLHttpRequest Call* (Chapter 6) to retrieve it. In this case, you'll probably want to keep the copy in a *Browser-Side Cache* for later use. Also, if you use an asynchronous call, which is advisable, there's a potential race condition: you need to ensure the stylesheet is retrieved before any transformation takes place, possibly with a loop that keeps checking for it. The later section "Code Refactoring: AjaxPatterns XSLT Drilldown Demo" demonstrates this approach.

Handcode the stylesheet as a JavaScript string

You can build up a string in JavaScript, though this leads to messy code that blurs the distinction between logic and presentation. There is one benefit of this approach: the stylesheet is dynamic, so could potentially vary according to the current context.

Store it inside the initial HTML document

You can tuck the stylesheet somewhere inside the initial HTML document where it won't be seen by the user. Techniques include:

- Make it the content of an invisible textarea.
- Store it in an *XML Data Island* the within document.

Real-World Examples

Google Maps

Google Maps (<http://maps.google.com>) is the most famous application of Browser-Side XSLT, where the technology is used to transform data such as addresses and coordinates into HTML. Based on this work, the Googlers have also released Google AJAXSLT (<http://goog-ajaxslt.sourceforge.net>), an open source framework for cross-browser XSLT and XPath.

Kupu

Kupu (<http://kupu.oscom.org/demo/>) is an online word processor (Figure 11-4) that stores content in XML and renders it with Browser-Side XSLT, using the Sarissa framework described shortly (Figure 11-4).

AJAX-S

Robert Nyman's AJAX-S (<http://www.robertnyman.com/ajax-s/>) is a slideshow manager, where raw content is maintained in XML, and transformed to slides using Browser-Side XSLT.

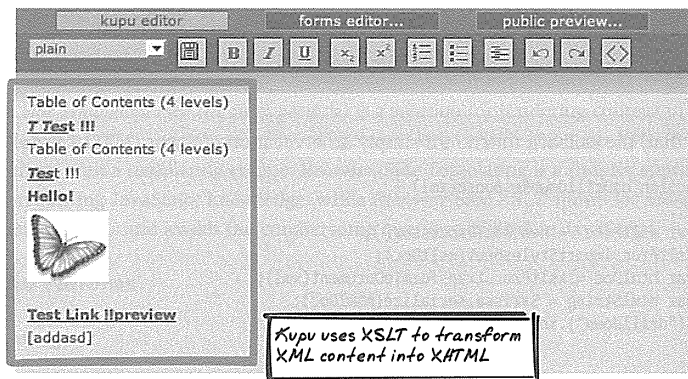


FIGURE 11-4. Kupu Word Processor

Sarissa framework

Sarissa (<http://sarissa.sourceforge.net/doc/>) is an open source, cross-browser framework for all things XML. XSLT is supported as well as XML parsing, XPath queries, and XMLHttpRequest invocation.

Code Refactoring: AjaxPatterns XSLT Drilldown Demo

This refactoring is similar to that performed in *Browser-Side Templating* but uses XSLT instead of templating. The starting point, the Basic Drilldown Demo (<http://ajaxify.com/run/portal/drilldown>), converts XML to HTML using a series of JavaScript string concatenations. The callback function, which receives the XML, therefore performs lots of string concatenations as follows:

```
html+="<div id='categoryName'>" + categoryName + "</div>";
```

In this refactoring, all that string handling is replaced by an XSLT transformation using the Sarissa library (<http://sarissa.sourceforge.net/doc/>). One slight complication is the stylesheet—how does the script access it? The solution here is to keep it as a separate file on the server and pull it down on page load:

```
var xsltDoc;
window.onload = function() {
    ...
    ajaxCaller.getXML("./drilldown.xml",function(response) {xsltDoc = response;});
    ...
}
```

As was mentioned earlier in “Decisions,” there’s a race condition here, because the XML response may come before the stylesheet. To deal with this, the drilldown callback function keeps looping until the stylesheet is defined:

```
function onDrilldownResponse(xml) {
    if (xsltDoc==null) {
```

```

        setTimeout(function() { onDrilldownResponse(xml); }, 1000);
        return;
    }
    ...
}

```

Beyond that, the callback function is simply an invocation of Sarissa's XSLT processor:

```

function onDrilldownResponse(xml) {
    ...
    var xsltProc = new XSLTProcessor();
    xsltProc.importStylesheet(xsltDoc);
    var htmlDoc = xsltProc.transformToDocument(xml);
    var htmlString = Sarissa.serialize(htmlDoc);
    $("drilldown").innerHTML = htmlString;
}

```

The only thing left is the stylesheet itself, shown below. I'll spare a walkthrough. The output here is the same as that in the Basic Drilldown Demo (<http://ajaxify.com/run/portal/drilldown>), which generates the HTML using manual JavaScript processing.

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <html>
            <body>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="/category">
        <div id='categoryName'><xsl:value-of select="@name" /></div>
        <xsl:if test="@parent!=''">
            <div id='parent' onclick='retrieveCategory("{@parent}")'>
                Back to <xsl:value-of select="@parent" />
            </div>
        </xsl:if>
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="/category/items/link">
        <div class="link"><a href="{url}"><xsl:value-of select=
"name"/></a></div>
    </xsl:template>

    <xsl:template match="/category/items/category">
        <div class="category"
            onclick="retrieveCategory('{@name}')">
            <xsl:value-of select="@name"/>
        </div>
    </xsl:template>

</xsl:stylesheet>

```

Alternatives

Browser-Side Templating

Browser-Side Templating (see the next pattern) is a solution for rendering content in any format—plain-text, XML, and so on. You could extract key variables from an *XML Message* and render them with a templating engine. Browser-Side Templating is a simpler approach as it builds on existing JavaScript knowledge, while Browser-Side XSLT allows for more powerful transformations and avoids the cumbersome querying of XML in JavaScript.

Related Patterns

XML Message

Browser-Side XSLT is driven by the need to deal with incoming *XML Messages* (Chapter 9).

XML Data Island

Browser-Side XSLT can be used to render an element contained in an *XML Data Island* (see earlier). Also, XML Data Island is a good place to stick an XSLT stylesheet.

Metaphor

Browser-Side XSLT is a strategy for presenting abstract data in a human-readable form. People do this all the time with diagrams. For instance, a family tree is one way to render the abstract set of relationships in a family. A UML diagram is a visual representation of some (real or imagined) code.

Browser-Side Templating



Pages, Presentation, Render, Template, Transform, View

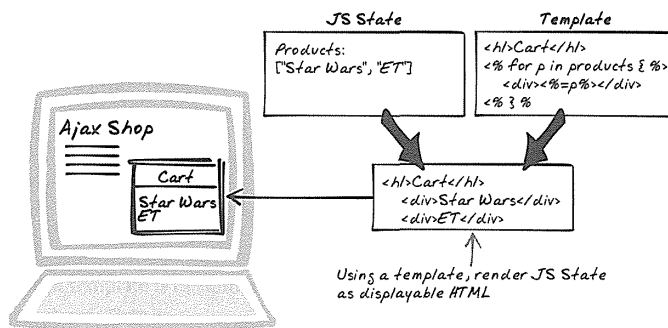


FIGURE 11-5. Browser-Side Templating

Developer Story

Devi has just been assigned to make the user profile more verbose. The original version takes an XML document about the user and renders it all with a jungle of JavaScript code that accumulates a big HTML string amid if-then conditions and loops. Devi decides the first step is to refactor all that into a Browser-Side Template, isolating the HTML generation. As a result, introducing the new content becomes trivial.

Problem

How can you separate presentation from logic?

Forces

- Generating HTML in the browser is a good way to isolate all presentation logic in one tier.
- To render context-specific information within the browser, you need to rely on a dynamic mechanism. It's not feasible to just set an element's `innerHTML` property to point to a static HTML page.
- Code gets complex and error-prone when you mix HTML generation with application logic. JavaScript is not well-suited to programmatically building up HTML strings.

Solution

Produce templates with embedded JavaScript and call on a browser-side framework to render them as HTML. The template contains standard HTML and allows context variables to be substituted in at rendering time, which gives it more flexibility than a static HTML page. In addition, you can also intersperse JavaScript code. For instance, generate an HTML table by running a loop—one iteration for each row.

The templating idea has been used for a long time on the Web, which is evident in frameworks like Perl's `HTML::Mason`, Java's JSPs, and the layout of languages like PHP and ASP. All of these are syntactic sugar of the "killer app" variety. That is, they technically don't add any new functionality, but make life a whole lot easier for web developers. In the Java world, the standard complement to JSPs is servlets. Here's how you'd write a message in a servlet:

```
package com.example.hello;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet {
```

```

protected void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Hi Everyboddeee!</title></head>");
    out.println("<body>");
    out.println("  Your name is <%context.name %>\\"" + request.getAttribute("name") +
        "\".");
    out.println("</body></html>");
    out.close();
}
}

```

Not pretty. There are printing commands all over the place, escaped special characters, quote marks, and error handling. What happened to the simplicity of HTML? Code like this is best avoided because it mixes presentation with logic. The following JSP, using the standard JSTL library, achieves the same thing in a more lucid, maintainable style:

```

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
<head><title>Hi Everyboddeee!</title></head>
<body>
  Your name is <c:out value="$name" />.
</body>
</html>

```

With many Ajax Apps, it's *deja vu*. Often, the browser receives a raw response such as a *Plain-Text Message* (Chapter 9) or an *XML Message* (Chapter 9) and must then render it as HTML, ruling out server-side templates. A common approach is to manually traverse the structure programmatically and create some HTML to be injected via some element's `innerHTML` property. The old "bucket of print statements" strikes again.

The solution here recapitulates in the browser what all those server-side templating frameworks offer. A templating framework, while not immensely difficult to implement, is probably not the sort of thing you'd want to write yourself. This pattern, then, is only made possible by the fact that frameworks already exist, and some are mentioned in the following examples. Here's a quick example of a template in Ajax Pages format (<http://ajax-pages.sourceforge.net/>):

```

<html>
<head><title>Hi Everyboddeee!</title></head>
<body>
  Your name is <%= name %>.
</body>

```

The JavaScript template is in similar format to the JSP above and follows standard conventions of templates. That is:

- Any code is contained inside `<%` and `%>` tags.
- Any expression to be substituted in the HTML is contained inside `<%=` and `%>` tags.

- To apply the template, create a processor and pass it a context. With Ajax Pages:

```
var ajp = new AjaxPages();
ajp.load("template.ajp");
var processor = ajp.getProcessor();
element.innerHTML = processor({name: username});
```

Decisions

How will you obtain the template?

You'll generally pass the template to the processor as a string, but where does that string come from? The options here are the same as with *Browser-Side XSLT* (see earlier in this chapter): store it server side; hardcode it as a JavaScript string; and store it inside the HTML document. See the "Decisions" section in *Browser-Side XSLT* for more details.

Will the template include any code? How much?

Code in templates is somewhat frowned upon because it defeats the purpose of templates as a place to isolate the presentation. Certainly, more complex calculation are best performed in plain JavaScript or on the server side, but if managing the display requires some logic, a template is a reasonable place to put it. Examples include:

Loops

A collection is passed in to the template. The template outputs the entire collection by looping over the collection. It's much better for the template to perform the looping, rather than calling JavaScript, because there is usually some HTML before and after the loop. If JavaScript outputted that HTML, there would be too much coupling between the JavaScript and the template.

Conditionals

if-then conditions and switch statements are sometimes better performed by the template too. However, if the body of each branch is long, you might prefer to include a different template for each.

How to prepare the template's context?

In the case of Ajax Pages (<http://ajax-pages.sourceforge.net/>), you're allowed to pass in a context variable at rendering time. That's usually a good thing, as it provides an alternative to the template using global variables. The most obvious thing to do is pass some existing objects to the template, but sometimes the calling code can prepare some extra information. The aim is to perform as much processing as possible in the JavaScript code, so as to avoid any complex logic being performed within the template. For example, sometimes the template contains a simple if-then condition like this:

```
<% if (context.credits == 0) { %>
  You need more credits!
<% } else { %>
  You have <%=context.credits%> credits!
<% } %>
```

The logic isn't especially complex, but scaling up with this approach can be problematic. As an alternative, let the JavaScript do some processing:

```
var creditsMessage = credits ?  
    "You need more credits!" : "You have " + credits + " credits!";
```

The template then gives you a clear picture of the HTML it will generate:

```
<%= context.creditsMessage %>
```

Real-World Examples

Ajax Pages framework

Gustavo Ribeiro Amigo's Ajax Pages (<http://ajax-pages.sourceforge.net/>) is an open source templating framework. There's a basic blog demo (<http://ajax-pages.sourceforge.net/examples/blog/index.html>) on the project homepage (Figure 11-6).

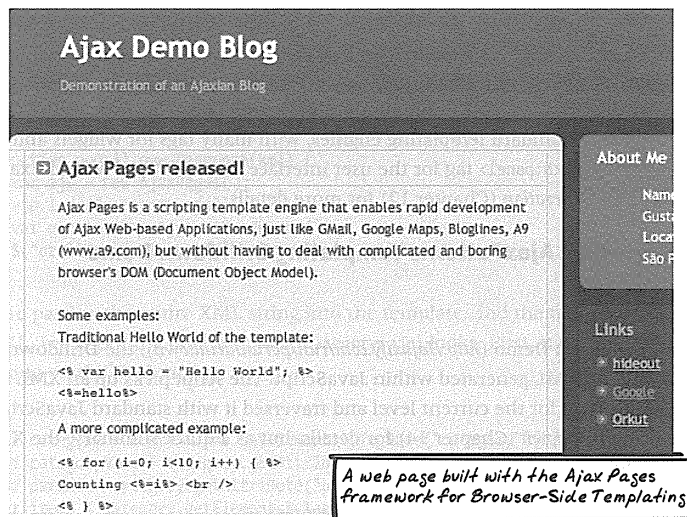


FIGURE 11-6. Ajax Pages demo

JavaScript Templates framework

JavaScript Templates (JST) (<http://trimpath.com/project/wiki/JavaScriptTemplates>) is an open source templating framework from TrimPath. JST offers a richer set of functionality at this time, including:

- Expression modifiers. The “capitalize” modifier, for instance, occurs in expressions like `${name|capitalize}`.
- A special syntax for loops and conditions.
- Macros.

These features may be useful if you're relying heavily on templating, but there's also an argument that JavaScript alone is sufficient, so there's no need to add the burden of learning a second set of similar syntax.

Here's how a JST template looks:

```
{for p in products}
  <tr>
    <td>${p.name|capitalize}</td><td>${p.desc}</td>
    <td>${p.price}</td>
    <td>${p.quantity} : ${p.alert|default:""|capitalize}</td>
  </tr>
{/for}
```

With the string in a DOM element on the web page, you can apply the template with a one-liner:

```
element.innerHTML = TrimPath.processDOMTemplate("templateElement", data);
```

Backbase framework

Backbase (<http://backbase.com/>) is a commercial framework that extends standard XHTML with Backbase-specific tags (BXML). It's a more over-arching framework, as the tags are more powerful than standard templating engines, with many tags for widgets and visual effects available; e.g., `<b:panel>` tag for the user interface. See the "Real-World Examples" in *Server-Side Code Generation* (Chapter 12) for more details.

Code Refactoring: AjaxPatterns Templating Drilldown Demo

Initial version

In the Basic Drilldown Demo (<http://ajaxify.com/run/portal/drilldown>), the Drilldown menu itself is a bunch of HTML generated within JavaScript. The script picks up an XML file containing a specification for the current level and traversed it with standard JavaScript and XML access. See *Drilldown* (Chapter 14) for details, but as a quick summary, the XML file looks like the following:

```
<category name="Overviews" parent="All Categories">
  <items>
    <link>
      <url>http://en.wikipedia.org/wiki/AJAX</url>
      <name>Wikipedia Article</name>
    </link>
    <link>
      <url>http://www.adaptivepath.com/publications/essays/archives/000385.php</url>
      <name>First Ajax</name>
    </link>
    <category name="Podcast Overviews" parent="Overviews" />
  </items>
</category>
```


The manual HTML generation looks like this:

```
function onDrilldownResponse(xml) {

    var category = xml.getElementsByTagName("category")[0];
    var html="";

    var categoryName = category.getAttribute("name");
    html+="

### Refactored to render from a template



For all the reasons just described, the HTML generation in the initial version is messy. The refactoring in this section introduces a template for the drilldown menu (http://ajaxify.com/run/portal/drilldown/template). Instead of the lengthy HTML generation in onDrilldownResponse(), the method becomes a simple application of an AjaxPages template:



```
function onDrilldownResponse(xml) {
 var ajp = new AjaxPages();
 ajp.load("category.ajp");
 var processor = ajp.getProcessor();
 $("drilldown").innerHTML = processor({xml: xml});
}
```



So we're passing the entire XML string into the template, and the template's converting it to HTML. Let's see how that template (category.ajp) looks. First, it performs a little pre-processing on the XML string:



```
<%
var category = context.xml.getElementsByTagName("category")[0];
var categoryName = category.getAttribute("name");
var parent = category.getAttribute("parent");
var items = category.getElementsByTagName("items")[0].childNodes;
%>
```



Then, it outputs the HTML. Note that looping and conditions are implemented with regular JavaScript, a fair reason to include JavaScript within a template.



```
<div id='categoryName'><%=categoryName%></div>

<%
if (parent && parent.length > 0) {
%>
 <div id='parent' onclick="retrieveCategory('<%=parent%>')">Back to

 <%=parent%></div>
<% } %>
<%
for (i=0; i<items.length; i++) {
```



DOM POPULATION 271



Amazon/DivX  
Exhibit 1028  
Page 289


```

```

        var item = items[i];
        if (item.nodeName=="link") {
            var name = item.getElementsByTagName("name")[0].firstChild.nodeValue;
            var url = item.getElementsByTagName("url")[0].firstChild.nodeValue;
%>
            <div class="link"><a href="<%=url%>"><%= name %></a></div>
<%
        } else if (item.nodeName=="category") {
            var name = item.getAttribute("name");
%>
            <div class='category'
                onclick="retrieveCategory('<%=name%>')"><%=name%></div>
<%
        }
    }
%>

```

We now have exactly the same external behavior as before, but the templating approach has helped separate presentation from logic.

Refactored to improve template context

In the previous version, the context consisted of only one thing: the entire XML string. As mentioned earlier in “Decisions,” it’s sometimes worthwhile doing some preparation before passing the context over to the template. In the preceding example, the template begins by extracting out a few convenience variables from the XML. That’s arguably okay, because it means the XML format is coupled only to the template, and not to the JavaScript. However, there’s also an argument that the JavaScript should simplify the template’s work by passing in a richer context. This further refactoring explores that avenue (<http://ajaxify.com/run/portal/drilldown/template/prepareContext>).

The change is quite small. The XML callback function now passes in a more detailed context:

```

function onDrilldownResponse(xml) {
    var ajp = new AjaxPages();
    ajp.load("category.ajp");
    var processor = ajp.getProcessor();
    var category = xml.getElementsByTagName("category")[0];
    $("drilldown").innerHTML = processor({
        categoryName: category.getAttribute("name"),
        parent: category.getAttribute("parent"),
        items: category.getElementsByTagName("items")[0].childNodes
    });
}

```

The template no longer needs the convenience variables as it can now refer to properties of the context:

```

<div id='categoryName'><%=context.categoryName%></div>

<%
    if (context.parent && context.parent.length > 0) {
%>

```

```

    <div id='parent' onclick="retrieveCategory('<%=context.parent%>')">Back to
    <br/>
    <%=context.parent%></div>
  <%
  }
  %>
  ...
  <%
    for (i=0; i<context.items.length; i++) {
    ...
    %>

```

Alternatives

Browser-Side XSLT

In many cases, the browser receives an XML response. Where the browser is converting XML to HTML, *Browser-Side XSLT* (see earlier) is a very direct alternative to this Browser-Side Templating. Templating simplifies presentation but still requires parsing of the XML, which can be cumbersome. XSLT simplifies the parsing as well, being designed specifically for the purpose of transforming XML.

Related Patterns

XML Message

Templating is well-suited to transforming an *XML Message* (Chapter 9) into HTML.

JSON Message

Templating is well-suited to transforming an object from a *JSON Message* (Chapter 9) into HTML.

Metaphor

Think of a physical template—a document with most content already present, with a few blanks to populate with current values.

Code Generation and Reuse

THERE ARE TWO GENERAL PROGRAMMING PATTERNS IN THIS CHAPTER. **SERVER-SIDE CODE GENERATION** IS one technique people are using to avoid working directly with HTML and JavaScript. *Cross-Browser Component* is a component that's been developed for portability across the major browsers.

Server-Side Code Generation



AutoGenerated, CodeGen, Framework, Library

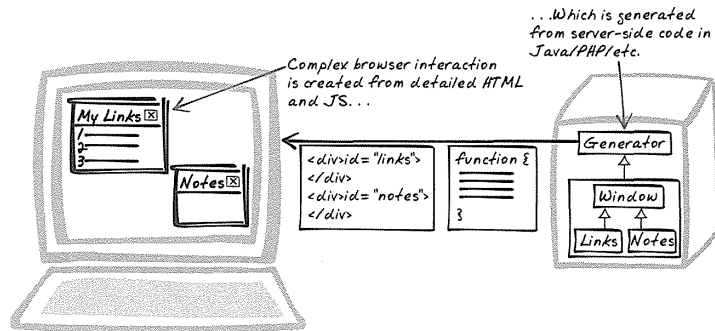


FIGURE 12-1. Server-Side Code Generation

Developer Story

Dave's creating a web app with a trendy new server-side framework, ZeroWeb. The framework lets him write everything in Java as if he's writing a standalone GUI application. It's capable of using that code to generate a conventional edition of the application, An Ajax edition, a desktop edition, and—for an extra \$99.95—a smartphone app too.

Problem

How do you avoid using HTML and JavaScript?

Forces

- HTML and JavaScript are the only code recognized by standard browser—any Ajax App must be represented in a combination of these formats.
- Ajax is increasing the complexity of HTML and JavaScript. Whereas JavaScript was previously used for simple tasks like navigation support or form validation, it's now driving the interaction in many applications.
- Most Ajax Apps also have a substantial backend, which is rarely implemented in JavaScript.
- It's difficult to completely separate the roles of browser-side developer and server-side developer. In fact, it's often more desirable to have the same developer working on a vertical slice of functionality; i.e., contributing a little to all tiers in order to deliver a new requirement. Thus, the same developer is often expected to work across tiers, where the environment differs significantly.
- Server-side development has certain advantages over JavaScript development. In some cases, there's better support for code editing, refactoring, testing, and so on.
- Sometimes, you want different user interfaces for the same application—a non-Ajax version or a desktop version, for example.

Solution

Automatically generate HTML and JavaScript from server-side code. You write code in a standard server-side language like Java and it's converted to HTML and JavaScript. Depending on the framework, you might not need to code directly in HTML or JavaScript at all. Since you're unlikely to create such a framework yourself, this pattern relies on a certain category of publicly available Ajax framework (see Appendix A).

In the case of all code being auto-generated, the code is something like a conventional desktop application. Widgets and event handlers are defined, and the browser generates the appropriate HTML, along with *XMLHttpRequest Calls* (Chapter 6) and general JavaScript handling. As well, the framework will often generate *Ajax Stubs* to deal with incoming calls and pass them to appropriate application code.

In other frameworks, the approach is more mixed. The framework might let you create a *Slider* with just a single tag, for example. But in the tag, you'll specify a JavaScript function to be called whenever the slider value changes.

There are several arguments for Server-Side Code Generation:

- It may be more productive, as the server-side language and framework may be more expressive and also because the framework might remove redundancy across the tiers (such as a data structure having a JavaScript representation as well as a server-side representation).
- Developers don't have to know HTML and JavaScript; they can work in the server-side language and environment they're most familiar with.
- The framework can help with graceful degradation. In other words, it can free the developer of worrying about browser portability, JavaScript, or cookies being turned off, or *XMLHttpRequest Calls* timing out.
- The framework can, in theory, deliver multiple platforms simultaneously. A single code base can be used to generate a web application (Ajaxian or not), standalone desktop application, and maybe even a mobile application.

However, code generation does come at a cost to usability. Look at it this way: anything you could generate from the server, a skilled developer could also hand code. The reverse is not true; given enough time and talent, it's always possible to build a custom solution that's superior in terms of usability. So, to be worthwhile, the framework must add enough value in terms of productivity and support for non-experts, to be compensated for the drop in usability. Whether this can be achieved really depends on two things: how much users will gain from a custom-built application, and how much of a boost the framework will give you. On the second point, consider a few counter-arguments to the benefits above:

- The first point, about developers not having to learn HTML and JavaScript, has traditionally held the most weight. Server-side developers in the past produced minimalist *satisficing* interfaces rather than *optimal* ones, allowing them to focus on the server side.

It's not uncommon to see advice against anything relatively complex being performed on the server side. But as Ajax becomes more prominent, this argument becomes less compelling. As discussed in *Fat Client* (Chapter 13), JavaScript is undergoing a renaissance—developers are learning more about it, tools and libraries are evolving, and patterns are emerging. All this reduces the argument for isolating developers from HTML and JavaScript.

- A server-side framework may well be able to handle graceful degradation and browser versions transparently. But to the extent that's possible in the server, a script can equally achieve the same effect in the browser, as discussed later in *Cross-Browser Component* (see later in this chapter). Portability is really a lame-duck argument for using server-side frameworks.
- The dream of transforming a single code base into a variety of different user-interface styles has been around for a while. Certainly, many modern UI toolkits are cross-platform, running on different operating systems from the same code base. However, can a single code base lead to both a web app and a desktop application? It's possible in theory and in proof-of-concept demos, but in practice, users expect applications to follow the conventions of the platform and that's difficult to achieve with a single generation.

An interesting variant of Server-Side Code Generation is “Browser-Side Code Generation”; the Backbase framework (<http://backbase.com>) is a case in point. With Backbase, the server outputs a specialized markup language, BXML, and the browser converts it into HTML, CSS, and JavaScript.

I expect the full-featured frameworks to become particularly prominent on intranet applications, where new functionality is often more critical than usability, and users can be trained to work around problems. Hand-coded HTML and JavaScript is more likely for public dot-coms where alternatives are abundant and the pressure of competition means it's critical to optimize the user interface. In those environments, there is still some role for Server-Side Code Generation, but mainly as a helper for very specific tasks and with some flexibility, rather than as the overarching architectural style.

Real-World Examples

These examples describe frameworks that support this pattern in different ways.

Echo2

Echo2 (<http://www.nextapp.com/products/echo2/>) is an open source framework by NextApp, which takes the “whole hog” approach and generates *all* HTML and JavaScript from Java source code. An online demo (<http://demo.nextapp.com/InteractiveTest/ia>) shows how Echo2's user-interface widget classes manifest themselves in the browser (Figure 12-2).

Ruby On Rails framework

Ruby On Rails (<http://rubyonrails.com>) is a web framework that embeds Ruby inside web pages. That in itself is not new, but Rails offers built-in helper classes to speed up page

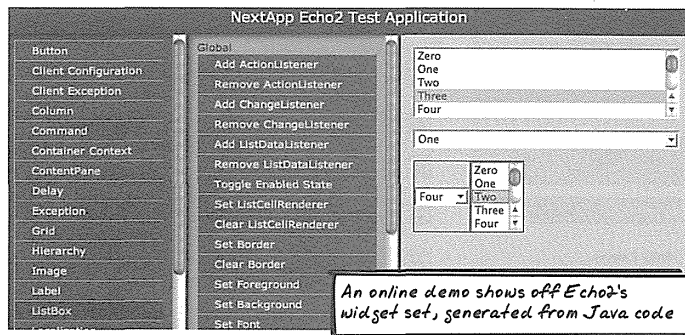


FIGURE 12-2. Echo2 demo

construction. They often lead to the Scriptaculous library being invoked, but the programmer doesn't use it directly. For example, you can issue an *XMLHttpRequest* Call followed by *Display Morphing*, using the following embedded Ruby call (http://www.onlamp.com/pub/a/onlamp/2005/06/09/rails_ajax.html):

```
<%= link_to_remote( "click here",
                  :update => "time_div",
                  :url => { :action => :say_when } ) %>
```

Backbase

Backbase (<http://backbase.com/>) is a commercial framework that extends standard XHTML with Backbase-specific tags (BXML). A Backbase client-side framework translates the BXML to HTML, CSS, and JavaScript, making it easy to implement many standard Ajax features. So there's a widget tag that looks like this:

```
<b:panel>
```

and another tag for various visual effects:

```
<s:fxstyle>
```

There are several innovative demos available from the Backbase homepage ([http://www.backbase.com/#home/pulldown/demos.xml\[2\]](http://www.backbase.com/#home/pulldown/demos.xml[2])) (Figure 12-3). On the server, Backbase offers programming interfaces for both .NET and JSF and offers visual development tools for Visual Studio .NET as well. As with the other tools here, the developer doesn't have to deal with HTML or JavaScript, but the framework also lets people customize the generation for specialized browser appearances and behavior. Note that, as mentioned earlier in the "Solution," Backbase is actually a kind of client-side, rather than server-side, code generation.

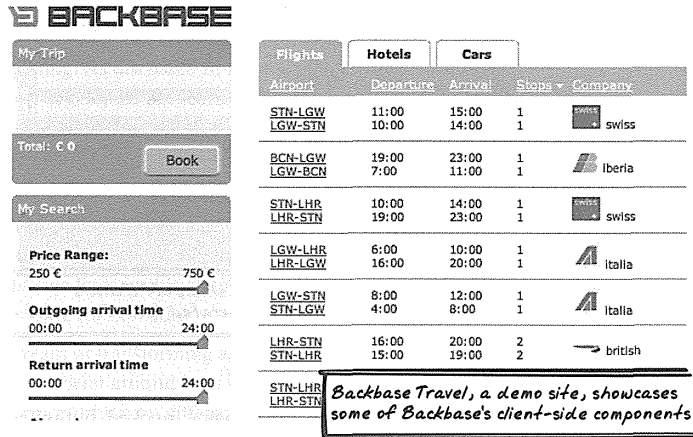


FIGURE 12-3. Backbase Travel demo

AjaxTags library

AjaxTags (<http://ajaxtags.sourceforge.net/>) is a collection of JSP tags to ease development of Ajax interaction. There's an <autocomplete> tag for *Suggestion* and an <ajax:update> field to perform a *Display Morphing* (Chapter 5) with the response of an *XMLHttpRequest Call* (Chapter 6).

Comfort.ASP library

Comfort.ASP (<http://www.daniel-zeiss.de/ComfortASP/>) takes an unusual approach by automatically adding Web Remoting (Chapter 6) to a conventional, non-Ajax application. By instrumenting the HTML, the framework can intercept standard form submissions and transform them into *XMLHttpRequest Calls*. By monitoring server output, Comfort's server-side script can respond to these calls by sending back only deltas—the difference between the current HTML and the desired HTML.

Code Example: Echo2 Number Guess

Echo2 (<http://www.nextapp.com/products/echo2/>) provides full source for a tutorial application, Number Guess (<http://www.nextapp.com/products/echo2/doc/tutorial/application.html>). The user guesses a number between 1 and 100, the application tells them if it's higher or lower, the user tries again, and so on. Note that the code is available, but there's no online demo at present.

An Echo2 application is run as a Java servlet extending Echo2's *WebContainerServlet*. The superclass takes care of most interaction; the concrete servlet must do just one thing: declare an *ApplicationInstance*. In Echo2, an *ApplicationInstance* object is the application state for a given user:

```

/**
 * Guess-a-number Tutorial Application Servlet Implementation.
 */
public class NumberGuessServlet extends WebContainerServlet {

    /**
     * @see nextapp.echo2.webcontainer.WebContainerServlet#newApplicationInstance()
     */
    public ApplicationInstance newApplicationInstance() {
        return new NumberGuessApp();
    }

}

```

The main application object initializes a new window and controls two top-level user-interface elements: a `GamePane` on startup, which is where the main action takes place, and a `CongratulationsPane` upon victory:*

```

public class NumberGuessApp extends ApplicationInstance {

    private Window mainWindow;

    public Window init() {
        mainWindow = new Window();
        mainWindow.setTitle("Echo2 Guess-A-Number");
        startNewGame();
        return mainWindow;
    }

    void startNewGame() {
        ...
        mainWindow.setContent(new GamePane());
    }

    void congratulate(int numberOfTries) {
        mainWindow.setContent(new CongratulationsPane(numberOfTries));
    }

}

class GamePane extends ContentPane implements ActionListener {
    ...
}

class CongratulationsPane extends ContentPane implements ActionListener {
    ...
}

```

The `GamePane` is built up by adding subelements and registering events. It acts as an `ActionListener` for the buttons it contains; hence, there's an `actionPerformed` event, which encapsulates the guess evaluation logic. The code illustrates that a framework like `Echo2` paves the way for a style of web programming that's almost identical to traditional desktop coding.

* All comments are omitted in the code sample.

```

class GamePane extends ContentPane implements ActionListener {

    private int randomNumber = ((int) Math.floor(Math.random() * 100)) + 1;
    private int lowerBound = 1;
    private int upperBound = 100;
    private int numberOfTries = 0;
    private TextField guessEntryField;
    private Label statusLabel = new Label();
    private Label countLabel = new Label("You have made no guesses.");
    private Label promptLabel= new Label("Guess a number between 1 and 100: ");
    private int guess;

    GamePane() {
        super();

        Column layoutColumn = new Column();
        layoutColumn.setCellSpacing(new Extent(10));
        add(layoutColumn);
        ...
        Button submitButton = new Button("Submit Your Guess");
        submitButton.setActionCommand("submit guess");
        submitButton.setForeground(Color.BLACK);
        submitButton.setBackground(Color.GREEN);
        submitButton.setWidth(new Extent(200));
        submitButton.addActionListener(this);
        layoutColumn.add(submitButton);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("new game")) {
            ((NumberGuessApp) ApplicationInstance.getActive()).startNewGame();
        } else if (e.getActionCommand().equals("submit guess")) {
            ++numberOfTries;
            ...
            if (guess == randomNumber) {
                ((NumberGuessApp) ApplicationInstance.getActive())
                    .congratulate(numberOfTries);
            } else if (guess < 1 || guess > 100) {
                statusLabel.setText("Your guess, " + guess + " was not between
                    1 and 100.");
            } else if (guess < randomNumber) {
                ...
            }
            ...
        }
    }
}

```

Alternatives

Browser-Side Templating

Browser-Side Templating (Chapter 11) is a form of Client-Side Code Generation, an alternative to Server-Side Code Generation.

Related Patterns

HTML Message

Server-Side Code Generation produces HTML responses, so is technically one form of *HTML Message* (Chapter 9). It's also a form of JavaScript Response, the variant described in *On-Demand JavaScript* (Chapter 6). However, the emphasis in those patterns is on directly created HTML and JavaScript, rather than delegation to a generation framework.

Cross-Browser Component



Agnostic, Common, Component, CrossBrowser, Independent, Portable

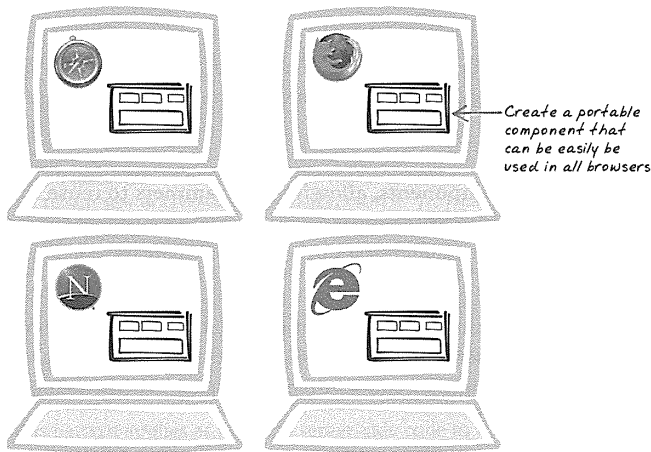


FIGURE 12-4. Cross-Browser Component

Developer Story

Dave has developed a cross-browser sprite engine, allowing users to easily drag a sprite around the screen. The library provides a browser-independent interface, taking into account browser incompatibilities regarding detection of mouse position and positioning of icons. When a new drag-and-drop requirement comes along, programmers can code to Dave's library in blissful ignorance of the underlying browser incompatibilities.

Problem

How can you manage cross-browser incompatibilities?

Forces

- A major selling point of Ajax is portability—the ability to run an application on any platform with a modern browser.
- There are numerous browsers on the market, and while IE still dominates, the market is becoming increasingly fragmented. IE currently has around 85–90 percent share on average web sites, but within certain user groups, Mozilla is actually far more popular. A recent survey of AjaxPatterns.org traffic showed 47.5 percent hits from Firefox, 30.1 percent from IE, and 2.7 percent from Safari.
- Browsers vary from each other in subtle and not-so-subtle ways. Sometimes, a browser simply doesn't support certain functionality. Other times, two browsers offer the same functionality, but it must be accessed in different ways. While the W3C (<http://www.w3c.org>) establishes standards for browsers, not all are followed.
- When creating a web program, you want to focus on business and application logic, without being distracted by idiosyncrasies of underlying browsers. Cross-browser compatibility is a cross-cutting concern, best dealt with as a separate task at a separate time.

Solution

Create Cross-Browser Components, allowing programmers to reuse them without regard for browser idiosyncrasies. A Cross-Browser Component abstracts away from underlying browser issues, offering a single API that can be used on any supported browser.

Following are a couple of examples of cross-browser incompatibilities. For simplicity, only the two most popular browsers are discussed: IE and Firefox. Other browsers are also important, and not surprisingly, introduce further incompatibility issues. The following are two examples:

XMLHttpRequest

XMLHttpRequest, a critical Ajax component, is constructed in a browser-specific way, as established in *XMLHttpRequest Call* (Chapter 6). Most browsers include a specific XMLHttpRequest class, but with IE prior to version 7, it's created as an ActiveX component instead. Furthermore, there are some subtle behavior differences. For example, XMLHttpRequest on Opera lagged behind in terms of support for features like authentication (<http://www.scss.com.au/family/andrew/webdesign/xmlhttprequest/>). An even greater problem is that many older browsers simply don't support this object, even though similar functionality can nonetheless be achieved using IFrames.

opacity

opacity—or transparency—is becoming important in Ajax applications, as discussed in *Popup* (Chapter 15) and in *One-Second Spotlight* (Chapter 16). However, portability is poor (<http://www.sitepoint.com/newsletter/viewissue.php?id=3&issue=102&format=html>). Recent versions of Firefox, Opera, and Safari allow for the CSS3 opacity style, older versions use their own different names, and IE uses a completely different approach based on a DirectX filter.

Any application using the features above must take into account cross-browser portability. Business and application logic is complex enough, without tangling it further with if-then portability statements. A better solution, and a common approach in the general area of portability, is to isolate cross-browser concerns.

Some browsers are available in multiple operating systems—Firefox, for example, runs in MS-Windows, Apple, Linux, and many other operating systems. It would be nice to assume a given browser was identical across all operating systems, but that's not the case. Some aspects of browsers rely on OS-specific features, which is where further incompatibilities arise. For example, IE's drag-and-drop capability not only depends on which version of IE is being used, but also on whether it's running on Apple or MS-Windows.

As a simple example, many libraries exist to make XMLHttpRequest calls portable. These libraries are examples of Cross-Browser Components, because the user doesn't have to worry about the browser being used. Most of these libraries internally use another form of Cross-Browser Component—a factory function that retrieves a suitable XMLHttpRequest object.

`ajaxCaller` (<http://ajaxify.com/run/testAjaxCaller/>), for example, contains the following factory function:

```
createXMLHttpRequest: function() {  
    try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}  
    try { return new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) {}  
    try { return new XMLHttpRequest(); } catch(e) {}  
    alert("XMLHttpRequest not supported");  
    return null;  
}
```

The function allows any code to create a new XMLHttpRequest object without worrying about browser specifics. The returned object is a Cross-Browser Component.

Note that this pattern applies as much to server-side code as to browser-side JavaScript. Whatever HTML comes from the server also needs to be portable. For that reason, you can also create server-side components that will generate portable HTML. So if you create custom JSP tags for your project, the JSP coder should be able to use them without having to make any browser-specific checks.

Decisions

What browser-specific criteria is used to create the Cross-Browser Component?

What criteria do you use to decide on a creation strategy? There are, broadly speaking, two approaches:

Version-dependent

Behavior is based on the browser version.

Feature-dependent

Behavior is based on checking for the existence of specific features—specifically whether a property or method exists.

The XMLHttpRequest example in the earlier “Solution” is an example of *feature-driven* behavior. Before trying to create a new window.XMLHttpRequest, we check if window.XMLHttpRequest exists. An equivalent version-dependent check would do something like this:

```
/* Version-Dependent Check (Unadvisable!!!) */
return determineIfCurrentBrowserIsIE() ?
    new ActiveXObject('Microsoft.XMLHTTP') : new XMLHttpRequest();
```

In the above example, a function determines if the current browser is IE. Based on that knowledge, the right mechanism can be used to create an XMLHttpRequest object. How might the version be determined? There are a few ways. First, the navigator object reports directly on this information. Useful properties include (<http://www.javascriptkit.com/jsref/navigator.shtml>):

```
navigator.appName
    Browser name; e.g. “Netscape”

navigator.appVersion
    Browser version; e.g., “5.0 (X11; en-US)”

navigator.UserAgent
    Detailed browser and platform details; e.g., “Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.9) Gecko/20050711 Firefox/1.0.5”
```

Ironically, the navigator object’s behavior is itself somewhat inconsistent across versions, so you need to handle strings like the userAgent differently. If you go this route, consider a browser detection library like TechPatterns’ JavaScript Browser & OS Detection (http://techpatterns.com/downloads/javascript_browser_detection.php) or Webreference’s JavaScript Browser Sniffer (<http://webreference.com/tools/browser/javascript.html>). Portability aside, the problem with direct interrogation is that the browser sometimes tells fibs—Opera, for example, lets users manually choose which browser to self-identify as, because some sites refuse to support browsers reporting as Opera.

An alternative way to detect browser version is to rely on a distinctive feature. For example, many scripts use document.all as a simple check for Internet Explorer (<http://weblogs.asp.net/jgalloway/archive/2005/08/07/421798.aspx>):

```
isIE=document.all;
```

Version-dependent behavior is feasible, quite common, and often recommended as the appropriate solution. One benefit over *feature-dependent* behavior is that you have only one or two variables to deal with; with feature-dependent behavior, you have to consider many combinations—what if the browser supports X but not Y?

However, there do remain some fundamental problems with any *feature-dependent behavior*. First, it’s cumbersome to track all the browser versions that support a particular feature. In many cases, you risk ruling out other browsers that may well support the feature. Also, version checks are often used in a binary way: either your browser’s supported and you can run the app, or it’s not supported and you’re out of luck. But if you focus on specific features, you can support *progressive degradation*: have a basic application that works

for all browsers, with certain additional features available only to those browsers that support them.

What if a feature isn't supported by the current browser?

Sometimes, a browser simply doesn't support a feature. What does a cross-browser API do when asked for functionality the browser doesn't provide? There are a couple of options:

Do nothing

Throw an exception or return null to indicate the object couldn't be created.

Provide a custom version

The browser may not support an object natively, but you might be able to create a custom version from other supported objects.

Real-World Examples

This section contains one example that showcases a particularly interesting type of cross-browser compatibility. For more examples of Cross-Browser Components, refer to Appendix A. Many of the JavaScript libraries mentioned there, like Dojo and Scriptaculous, aim to provide components that work on all major browsers.

HTMLHttpRequest library

Angus Turnbull's HTMLHttpRequest (<http://www.twinhelix.com/javascript/htmlhttprequest/>) is a cross-browser library for Web Remoting (Chapter 6). Unlike many of the recent Ajax libraries, it's able to gracefully degrade to IFrame usage when executed on older browsers. The API's still the same, so a programmer can access web remoting in the same way, regardless of the browser in question.

Code Example: HTMLHttpRequest Library

HTMLHttpRequest (<http://www.twinhelix.com/javascript/htmlhttprequest/>) is a cross-browser remoting component. It works by wrapping around a real, concrete remoting object—either an XMLHttpRequest object or an IFrame, only one of which will be set. The construction code below makes extensive use of feature-dependent behavior to decide, at each stage, whether to proceed with a particular strategy. There is also some use of feature-dependent behavior to ensure that Opera and IE5 don't follow a particular path.

```
function HTMLHttpRequest(myName, callback){
    this.myName=myName;
    this.callback=callback;
    this.xmlhttp=null;
    this.iframe=null;
    ...
    if(window.XMLHttpRequest){
        xmlhttp=new XMLHttpRequest();
        if(xmlhttp.overrideMimeType)
            xmlhttp.overrideMimeType('text/xml')
    }
    if(!xmlhttp) {
```

```

        if( document.createElement && document.documentElement
            && (window.opera||navigator.userAgent.indexOf('MSIE 5.0')== -1)) {
            ...
        } else if (document.body && document.body.insertAdjacentHTML) {
            ...
        }
        ...
    }
    return this;
}

```

Related Patterns

Server-Side Code Generation

Server-Side Code Generation (see earlier) is often a convenient way to generate Cross-Browser Components, allowing you to perform all the abstraction logic server side.

On-Demand JavaScript

If you design carefully, you can separate out the implementations for each platform. Use the *Lazy Loading* technique of *On-Demand JavaScript* (Chapter 6) to ensure that only code required for the user's own platform is downloaded.

Metaphor

Using a Cross-Browser Component is like working with a certified professional. The professional may have gained their certification in different ways and might offer different services, but there remains a basic set of services that you can expect to be carried out adequately.

Want to Know More?

- Good discussion on cross-browser issues and specifically enhancing IE applications to support Mozilla by Doron Rosenberg (<http://www-128.ibm.com/developerworks/web/library/wa-ie2mozgd/>)
- Quirksmode, by Peter-Paul Koch, is a web site providing comprehensive compatibility resources (<http://quirksmode.com>)

Performance Optimization

THE PERFORMANCE OPTIMIZATION PATTERNS IMPROVE THROUGHPUT AS WELL AS APPLY SOME USER-centered techniques to ease the pain of waiting for the server. *Browser-Side Cache* talks about implementing a custom cache within the application and explains why it's sometimes more suitable than the standard web browser cache. *Predictive Fetch* extends the cache concept by proposing that likely actions be anticipated, so that required data is already sitting in the browser by the time it's required. Another very different way to achieve zero network delay is to simply take a stab at the required value, the idea behind *Guesstimate*.

Multi-Stage Download proposes parallel or scheduled downloads so as to reduce bottlenecks and grab the most critical details as early as possible.

Fat Client is one of several possible Ajax architectural styles that helps optimize performance by pushing as much logic as possible—and potentially storage—over to the browser side.

Browser-Side Cache



Auto-Update, Memoise, Memoize, Sync, Synchronise, Synchronize, Real-Time

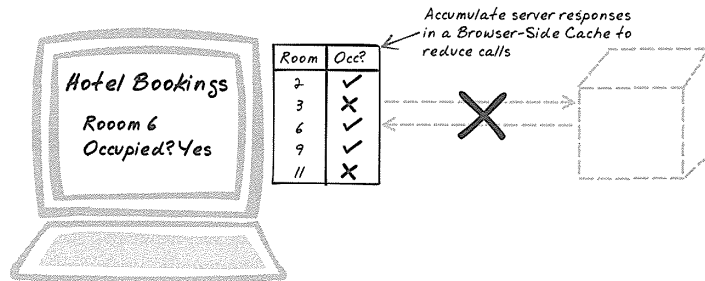


FIGURE 13-1. Browser-Side Cache

Developer Story

Devi's produced a physics simulation and, though it's functional, she's disappointed that it keeps pausing to request calculations from the server. To speed it up, she sets up a cache that will retain the calculations locally.

Problem

How can you make the system respond quickly to user activity?

Forces

- The application should respond to user actions quickly—ideally, instantaneously.
- Many user actions require a response from the server.
- Responses from the server can be noticeably latent due to data transfer and server processing overheads.

Solution

Retain server results in a Browser-Side Cache. The cache is a JavaScript map-style object that holds query-result pairs; the queries are cache keys and the server results are server results. So whenever the browser needs to query the server, first check the cache. If the query is held as a key in the cache, the corresponding value is used as the result, and there's no need to access the server. Caching like this is performed in many systems, but there are some interesting Ajax-specific issues, such as the relationship with the browser's built-in cache and the asynchronous nature of calls.

XMLHttpRequest Call (Chapter 6) explains that call results can already be cached by the browser, so you might wonder why this pattern exists. After all, the browser not only handles caching for you, but can cache huge quantities of data using the filesystem, and,

unlike a Browser-Side Cache, this data will usually live beyond the current session. The reason you'd use a Browser-Side Cache is to exert more control over caching. You get to choose how much data is cached and how long it lasts, whereas those things are set by the user in a web browser cache. Moreover, you bypass the notorious portability issues associated with cache control, choosing exactly what will be and what will not be cached. Another advantage of a Browser-Side Cache over the web browser's cache is that you can save JavaScript objects you've built up, thus saving the effort of reconstructing them later on (a process known as "memoisation").

What exactly is the format of the key and value? In the simplest case, the query would just be the URL, and the result would be the response body. However, that's not so useful because if you make calls correctly, as discussed in *XMLHttpRequest Call* (Chapter 6) and *RESTful Service* (Chapter 9), the browser will handle caching on your behalf, and as it's backed by the filesystem, it can cache much greater quantities of data. However, more useful are caches that hold high-level semantic content—typically the results of processing server responses.

A cache data structure requires some way to access the key-value pairs "randomly"—that is, directly, without having to traverse the entire data structure until the desired key-value pair is found. In JavaScript, a cache can be created using an associative array (in JavaScript, this is the same as an object). So an empty array is created at startup and gradually accumulates server results. Of course, an Ajax App lives indefinitely, so the array could continue growing if not controlled. There are a few ways to handle this, as discussed later in "Decisions."

It's common in many systems to treat the cache as a Proxy, as shown in Figure 13-2 (see Gamma et al.'s [1995] Proxy pattern). That is, clients retrieve results from an object called something like "ResultFetcher," which encapsulates anything to do with fetching of results. From their perspective, the ResultFetcher just provides results. It doesn't matter to the client whether all results are directly pulled from the server or whether some of them are cached inside ResultFetcher.

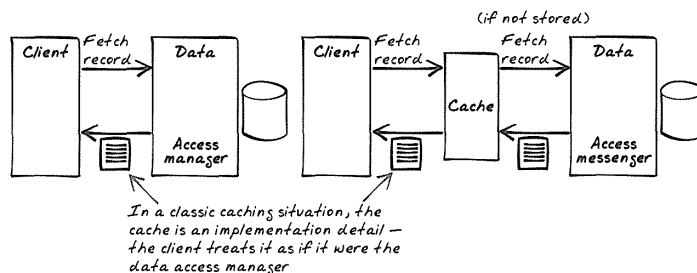


FIGURE 13-2. Standard synchronous caching setup

Here's where the asynchronous nature of XMLHttpRequest adds a slight complication. Under normal circumstances, a caching ResultFetcher always returns the desired value

synchronously, whether the result was in the cache or not. But with Ajax, it can only do this if the result was indeed found in the cache. Otherwise, the server response will come back some time later, after the fetching function has already returned. To deal with this, you might consider having the requester pass a callback function to the fetching procedure. So the fetching procedure never returns anything directly, but instead ensures the callback function is eventually called with the desired value. If it knows the value immediately, the callback function will be called straightaway, and if not, it will be called later on. In other words, the ResultFetcher has roughly the same interface and external behavior as XMLHttpRequest (or a suitable wrapper).

Decisions

What will be stored as keys? For values?

The solution mentioned a couple of possibilities for keys and values. In the first instance, they might simply be URLs and response bodies. Using the entire URL and response body does come at a cost, however. Both contain a lot of useless information, which limits how many items can be stored in the cache. An alternative is to use semantically related values. For example, the key might be customer names, and the values might be customer objects. Semantic values also lets you incorporate other data, such as results from calculations.

How will cache size be kept under control?

You typically keep the cache size under control by deciding on its capacity and on some way to remove elements when that size has been reached. Each new element usually results in an older element being discarded. For efficiency, you might discard a large batch at once, and then let the cache gradually build up again.

Two common algorithms are (http://en.wikipedia.org/wiki/Cache_algorithms):

Least Recently Used (LRU)

The discarded item is the one with the longest time since it was last retrieved.

Least Frequently Used (LFU)

The discarded item is the one that has been retrieved the least.

Both algorithms are feasible in JavaScript, provided you use the right data structure. “Code Example: Cached Sum Demo,” later, illustrates LRU.

How will you protect against stale data?

Regarding stale data, the first question to ask is, “How recent does the data have to be?” If it’s real-time stats being used by a nurse to monitor a patient’s health, it probably needs to be pretty recent. So much so, that a cache might even be out of the question. If it’s a student perusing a 50-year old article on ancient Athenian literature, a 12-hour cache will do fine.

There are several ways to enforce this decision:

- Attach a timestamp to each item that goes into the cache. Whenever you retrieve an item, inspect the timestamp to determine if the result is recent enough.
- As a variant on the above, schedule a periodic loop to actively delete stale items.
- Implement a browser-server protocol that allows the browser to determine if items are stale. So the browser might keep a timestamp for each item, and the server exposes a service to accept a timestamp. The server only needs to send the whole response if the value has changed since that time. An alternative to timestamps would be a hash function—a function that the browser runs against the cached value that can then be compared by the server against the item’s most recent hash value, to see if a change has occurred.
- Implement a service on the server that announces changes that have occurred. Use *Periodic Refresh* (Chapter 10) to actively listen for such changes and to delete from the cache any items that have changed.

Real-World Examples

This section contains a *XMLHttpRequest Call* library with caching support. “Real-World Examples” in *Predictive Fetch* includes several other systems with Browser-Side Caches.

libXmlRequest library

Stephen W. Cote’s libXmlRequest (<http://www.whitefrost.com/servlet/connector?file=reference/2003/06/17/libXmlRequest.html>) was one of the earlier wrappers of the XMLHttpRequest object. A typical asynchronous request looks like this:

```
getXml(path, callbackFunction, 1, requestId);
```

To make the request cached, simply add an extra argument at the end:

```
getXml(path, callbackFunction, 1, requestId, 1);
```

The approach shows how cache functionality can be made orthogonal to core application functionality.

Code Example: Cached Sum Demo

The Basic Sum Demo (<http://ajaxify.com/run/sum/>) has to resort to the server each time the cache is reached. This refactoring adds caching functionality to reduce server calls and is implemented in three stages.

Including the query in the response

The asynchronous nature of *XMLHttpRequest Call* separates the initial query from the response, so it’s not always clear when a request comes in what the corresponding request was. One way to achieve this is to include the original query as part of the response. That’s the first refactoring here. The refactoring is detailed in *XML Message* (Chapter 9), and the net effect is a response like the one that is shown next (<http://ajaxify.com/run/sum/xml/sum.XML.php?figure1=4&figure2=8&figure3=>).

```

<sum>
  <inputs>
    <figure id="1">4</figure>
    <figure id="2">8</figure>
    <figure id="3"></figure>
  </inputs>
  <outputs>12</outputs>
</sum>

```

Previously, the response was just 12. The resulting value is extracted from the XML in a slightly different way, but the inputs are not used until the next iteration.

An infinite cache

The next refactoring creates a very basic cache—one that has no regard for the laws of physics—it just keeps growing indefinitely. That’s a bad thing, but a useful stepping stone to the final iteration. The cache holds the sum against a `figuresString`, simply a comma-separated list of the three figures.

First, the cache is created by a global variable and initialized from the `onload` method:

```

var sumByFigures;
...
function restartCache(html) {
  sumByFigures = new Array();
  ...
}

```

Each time a sum is submitted, `figuresString` is calculated to form a key for the cache. The cache is then interrogated to see if it already contains that sum. If not, an asynchronous call is set up. Either way, the ultimate consequence is that `repaintSum()` will eventually be called with the new sum. If the result is already cached, it will be called straightaway. If not, it will be called after the server has returned.

```

function submitSum() {
  ...
  var figuresString =
    figures.figure1 + "," + figures.figure2 + "," + figures.figure3;
  var cachedSum = sumByFigures[figuresString];
  if (cachedSum) {
    repaintSum(cachedSum);
  } else {
    repaintSum("---");
    ajaxCaller.get("sumXML.php", figures, onSumResponse, true, null);
  }
}

```

`onSumResponse` not only calls `repaintSum()` with the value in the response, but also pushes the result onto the cache:

```

function onSumResponse(xml, headers, callingContext) {
  var sum = xml.getElementsByTagName("output")[0].firstChild.nodeValue;
  repaintSum(sum);

  var figures = xml.getElementsByTagName("figure");

```



```

    var figuresString = figures[0].firstChild.nodeValue + ","
                      + figures[1].firstChild.nodeValue + ","
                      + figures[2].firstChild.nodeValue;
    sumByFigures[figuresString] = sum;
}

```

Finally, `repaintSum` is the function that detects a change—either way—and simply morphs the display:

```

function repaintSum(html) {
    self.$("sum").innerHTML = html;
}

```

A finite cache

The final cache (<http://www.ajaxify.com/run/sum/xml/cached/expiry/>) enhances the previous version by introducing a least-recently used disposal algorithm. Each time a new item is added to the cache, the least recently used item is discarded from the cache. It would be inefficient to trawl through the entire cache each time that happens, comparing usage times. So, in addition to the associative array, a parallel data structure is composed. It's a queue, where each new item is pushed to the tail of the queue and gradually approaches the head as further items are pushed on. When the queue is full and an item is pushed on to the tail, each item moves down one, and the head item "falls off" the queue, so it is deleted from both the queue and the associative array. Whenever an item is retrieved, it's sent back to the tail of the queue. That's what ensures the least recently used item is always at the head.

The queue itself is a class with the following functions: `enqueue()`, `dequeue()`, and `sendToTail()`. It works by tracking the head, the tail, and the size, and by keeping the items in a doubly linked list. For example, `enqueue()` is defined like this:

```

Queue.prototype.enqueue = function(obj) {
    newEntry = {
        value: obj,
        next: this.tail
    }
    if (this.tail) {
        this.tail.prev = newEntry;
    } else { // Empty queue
        this.head = newEntry;
    }
    this.tail = newEntry;
    this.size++;
}

```

Back to the sum application, which now declares a queue as well as the associative array:

```

var figuresQueue, sumByFigures;

```

Each time a new element arrives from the server, it's sent to `encache()`. `encache` will lop the least recently used item off both data structures if the queue is full. Then it will add the new item to both.

```

function encache(figuresString, sum) {
    // Add to both cache and queue.
    // Before adding to queue, take out queue head and
    // also remove it from the cache.
    if (figuresQueue.size == cacheSize) {
        removedFigures = figuresQueue.dequeue(figuresString);
        delete figuresString[removedFigures];
    }
    figuresQueue.enqueue(figuresString);
    sumByFigures[figuresString] = sum;
    $("#queueSummary").innerHTML = figuresQueue.describe();
}

```

Whenever the cache is queried, the queried value is not only returned, but is also sent back to the tail of the queue to mark it as having been recently used:

```

function queryCache(figuresString) {
    // Recently used, so move corresponding entry back to tail of queue
    // if it exists.
    figuresQueue.sendToTail(figuresString);
    $("#queueSummary").innerHTML = figuresQueue.describe();
    return sumByFigures[figuresString];
}

```

With these abstractions in place, there is not much change to the core part of the sum script. `submitSum()` queries the cache and calls the server if the result is not found. And the server response handler ensures new results are added to the cache:

```

function submitSum() {
    ...
    var cachedSum = queryCache(figuresString);
    if (cachedSum) {
        repaintSum(cachedSum);
    } else {
        repaintSum("---");
        ajaxCaller.get("sumXML.php", figures, onSumResponse, true, null);
    }
}
...
function onSumResponse(xml, headers, callingContext) {
    ...
    encache(figuresString, sum);
}

```

Alternatives

Built-in browser cache

As explained in the solution, the built-in browser cache is an alternative form of cache. It's better suited for larger data HTML, but is more difficult to control than a Browser-Side Cache.

Server-side cache

Caching data on the server cuts down on processing, especially when the data is shared by multiple users. However, that’s mainly a benefit if the server processing is the bottleneck. Bandwidth is more often the chief constraint, and the server-side cache won’t reduce browser-server traffic. The best option is often a combination of browser-side and server-side caching.

Related Patterns

Submission Throttling

Browser-Side Cache focuses on “read caching,” in which responses from the server are held in the cache. Another flavor of caching is “write caching,” where output is held in a buffer to defer outputting to its true destination. *Submission Throttling* (Chapter 10) is a form of write-caching.

Predictive Fetch



Anticipate, Fetch, Guess, Prefetch, Preload, Prepare, Ready

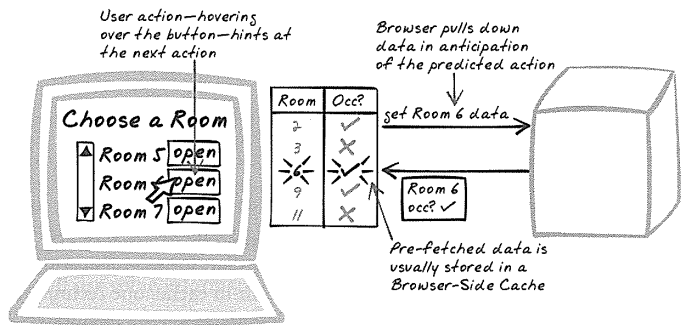


FIGURE 13-3. Predictive Fetch

Developer Story

Dave’s tuning a music web site and the logs tell him that users who listen to a song for more than a minute are quite likely to click the Purchase button. So he introduces a new timer—at the one-minute mark of each song, the browser will quietly download pricing data just in case the user wants to buy it later on.

Problem

How can you make the system respond quickly to user activity?

Forces

- The application should respond to user actions quickly; ideally, it should feel instantaneous.
- Many user actions require a response from the server.
- Responses from the server can be noticeably latent due to data transfer and server processing overheads.

Solution

Pre-fetch content in anticipation of likely user actions. Pre-fetching attempts to remove the delay altogether for certain user actions.

The obvious motivation for instantaneous feedback is efficiency: the things can happen faster because the user's not sitting around waiting for results. In reality, though, the proportion of overall time waiting might actually be quite small. The more severe problem is distraction, because delays will break the user's concentration, along with the sense of frustration at not being in control. The other problem is that for real-time applications, the user is slower to respond to state conveyed by the server. A chat session will be more strained, and a remote device under browser control will be more erratic.

Here are some occasions when Predictive Fetch might be used:

- The user's navigating a *Virtual Workspace* (Chapter 15) such as a large table. Pre-fetch the results of moving in each direction.
- The user's converting between two currencies. Pre-fetch the major currency rates.
- The user's reading some articles. Pre-fetch all stories in her favorite category.

Predictive Fetch usually requires a *Browser-Side Cache* (earlier in this chapter) in order to accumulate the pre-fetched content. In addition, you can exploit the web browser's built-in cache in a couple of ways—see *Browser-Side Cache*, earlier, for a comparison of these techniques. To get something in the web browser's built-in cache, one technique is to issue *XMLHttpRequest Calls* for content you'll need later on, using the techniques mentioned in that pattern to encourage the content to be cached. The response handler does nothing here—the point is simply to bring the content into the browser, where it will be cached for later on. The other technique is for images: create a dummy image (`document.createElement("img")`) object and set its `src` property to whatever image URL you want to preload. Again, we're only doing this to ensure the image goes in the cache, so the image isn't actually attached to the page.

It's rarely possible to pre-fetch for *all* actions, so the designer has to be smart about anticipating which actions are most likely. Higher priority will be given to content that is more important or more likely to be used.

One problem with pre-fetching is the application will be a bit erratic. The user might be surprised that some commands respond instantaneously, while similar commands take a long time. While some variation is always expected on the Web, the main problem comes

when the data arrives immediately. The user will reasonably question whether the app was really taking their action into account (<http://www.baekdal.com/articles/Usability/usable-XMLHttpRequest/>). For that reason, a useful but counter-intuitive trick is to actually fake a small delay and use a visual effect like *One-Second Spotlight* (Chapter 16) to hint that the server really was involved.

Decisions

How much information will be pre-fetched?

Pre-fetching comes at a cost. Anticipating the user's actions is a guessing game, and for each guess that goes wrong, some resources have been wasted. Designers must make a trade-off involving the likelihood that pre-fetched data will be used, the user benefit if it is used, and the overhead if it's not used. This could involve some fairly heavy user analysis combined with statistical methods.

In practice, it's feasible to use some initial rules of thumb and proceed on a more empirical basis. With the right design, it should be easy enough to discriminately turn pre-fetching on and off. Thus, by studying logs and comparing the effects of pre-fetching different aspects, it's possible to evolve the algorithms being used.

Will it be the server or the browser that anticipates user actions?

The request for information will always come from the browser, but it's feasible for either the server or the browser to anticipate what the user will need next. If the browser is to decide, it can simply issue a request for that information. If the server is to decide, the browser can, for example, issue a periodic request for general information—perhaps with some indication of current state—and the server can then push down whatever information it decides might come in handy for the browser.

What information can be used to anticipate user actions?

It's likely you'll use some information about the present and the past to help predict the future. There's a lot of data you could potentially use to help decide what the user will do next:

User's profile

The user's profile and history should provide strong cues. If the user always visits the Books area as soon as they log on, then the homepage should pull down Books content.

User's current activity

It's often possible to predict what the user will do next from their current activity. If a user has just added an item to his shopping cart, he will likely be conducting a purchase in the not-too-distant future; consider downloading his most recent delivery address.

Activity of other users

Sometimes, a rush of users will do the same thing at once. If the user has just logged into a news portal while a major news event is in progress, system-wide statistics will inform the server that this user is probably about to click on a particular article.

Collaborative filtering

As an extension of the previous point, a more sophisticated technique is to correlate users based on information such as their profile and history. People are likely to behave similarly to those whose details are highly correlated. So if a user tends to look at “Sport” followed by “Weather,” then the system should start pre-fetching “Weather” while the user’s looking at “Sport.”

Real-World Examples

Google Maps

Some experimentation with Google Maps (<http://maps.google.com>) suggests Predictive Fetch is used while navigating the map (Figure 13-4). The evidence is that you can slowly move along the map in any direction, and you won’t see any part of the page going blank and refreshing itself. If you move quickly, you’ll see the refresh behavior. Based on that observation, the map is apparently fetching content beyond the viewable area, in anticipation of further navigation.

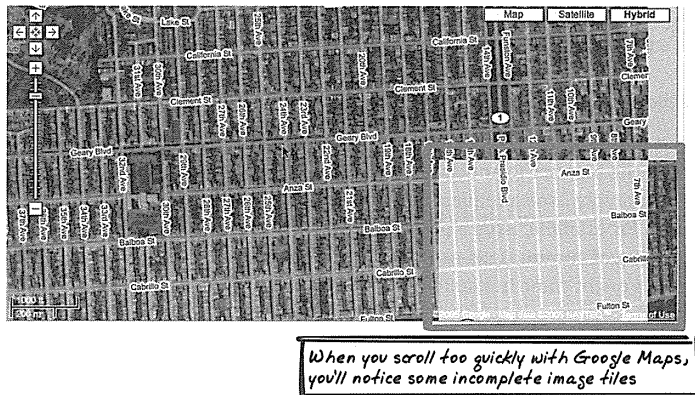


FIGURE 13-4. Google Maps loading

map.search.ch

map.search.ch (<http://map.search.ch>) includes familiar buttons for zooming and panning. The difference is that as soon as a mouse pointer hovers over a button, a call is made to anticipate the user clicking on it. This is a good compromise between downloading in every possible direction and zoom level or doing nothing at all.

Firefox “Prefetch”

This is not entirely an Ajax example, but it’s worthwhile noting a particular Firefox (and Mozilla) feature called “prefetching.” The HTTP protocol allows for new link types to be

defined, and Firefox happens to define a “Prefetch” link type (http://www.mozilla.org/projects/netlib/Link_Prefetching_FAQ.html#What_is_link_prefetching). A “prefetch” link looks like this:

```
<link rel="prefetch" href="/images/big.jpeg">
```

When Firefox sees such a link appear, it will generally fetch the associated content, which is ready to be shown immediately. An application can exploit that feature by including links to content that is likely soon to be requested.

Google Search, for example, slaps a prefetch directive around the first search result, but states that this occurs for “some searches” only. I am guessing, based on some experimentation, that this means searches where Google is highly confident the top search will be chosen. So a search for the highly ambiguous term “anything,” results in no prefetch directive. Meanwhile, a search for IBM, where it’s obvious what most users seek, will direct Firefox to prefetch the first result:

```
<link rel="prefetch" href="http://www.ibm.com/">
```

International Herald Tribune

Another example is The *International Herald Tribune* (<http://www.iht.com/>), which caches entire articles to provide instant gratification when you click on Next Page.*

Code Refactoring: AjaxPatterns Predictive Fetch Sum

The Basic Sum Demo (<http://ajaxify.com/run/sum/>) allows the user to enter a set of figures, but he must wait for the server in order to see the sum. The Cached Sum Demo (<http://ajaxify.com/run/sum/xml/cached/>) improves on the situation a bit, by ensuring recent sums do not need to be re-fetched. However, there is still no anticipation of user behavior. In this refactoring (<http://ajaxify.com/run/sum/predictive/>), a new sum is fetched each time the user types something. By the time he has pressed the Add button, the browser should hopefully have the result waiting in the cache, ready to be displayed immediately. Each time a new sum is anticipated, the browser fetches the result, and a callback pushes it into the cache.

The code change is actually fairly straightforward. As before, clicking on the Add button causes a server sum request. This time, though, a flag is set to indicate its a real user submission as opposed to a pre-fetch. The call will “remember” the flag; i.e., using `ajaxCaller`’s `callingContext`, the flag will be passed to the callback function when the response is issued:

```
$("#addButton").onclick = function() {
    submitSum(true);
}
```

For Predictive Fetch, each release of a key triggers a sum submission, with the callback false to indicate this was *not* a user submission.

* This example is taken from an O’Reilly Network article (<http://www.oreillynet.com/pub/wlg/6782>).

```

$("figure1").onkeyup = function() { submitSum(false); }
$("figure2").onkeyup = function() { submitSum(false); }
$("figure3").onkeyup = function() { submitSum(false); }

```

`submitSum()` will perform the same request as before, but this time passes in the flag as a calling context:

```

ajaxCaller.get("sumXML.php", figures, onSumResponse, true, isUserSubmitted);

```

The `isUserSubmitted` flag is then retrieved by the callback function, which will only change the display if the corresponding query was user-submitted. Either way, the result is then pushed into the cache for later retrieval:

```

function onSumResponse(xml, headers, callingContext) {
    var isUserSubmitted = callingContext;
    ...
    if (isUserSubmitted) {
        repaintSum(sum);
    }
    ...
}

```

Alternatives

Fat Client

Sometimes, the server is invoked to perform some logical processing, rather than to access server-side data. If that's the case, it may be possible to process in JavaScript instead, as suggested by *Fat Client* (see later).

Server priming

In some cases, the reverse may be more appropriate: it might make sense for the browser to hint to the server what the user is working on, without the server actually responding. How is this useful? It allows the server to get started on processing, so that the required information can be cached server side but not actually pushed into the browser's own cache.

Related Patterns

Guesstimate

Guesstimate (see the next pattern) is another performance optimization based on probabilistic assumptions. A *Guesstimate* involves the browser guessing the current server state, whereas a *Predictive Fetch* involves guessing what the user will do next.

Browser-Side Cache

Browser-Side Cache (see earlier) is a prerequisite to *Predictive Fetch*—the predictively fetched data has to be stored in some form of cache.

Metaphor

Time-shifted media—evident in Tivo, offline RSS readers, and podcatchers—is an example of Predictive Fetch.

Guesstimate



Approximate, Estimate, Extrapolate, Fuzzy, Guess, Guesstimate, Interpolate, Predict, Probabilistic, Sloppy, Trend

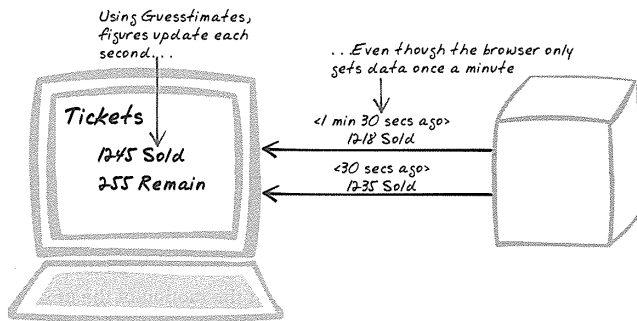


FIGURE 13-5. Guesstimate

Developer Story

Devi's producing a taxi tracker so the head office knows where its fleet is at any time. The taxis only transmit their location every 10 seconds, which would ordinarily lead to jerky movements on the map. However, Devi wants the motion to be smooth, so she uses interpolation to guess the taxi's location between updates.

Problem

How can you cut down on calls to the server?

Forces

- To comprehend system activity and predict what might happen next, it's useful to have frequent updates from the server.
- It's expensive to keep updating from the server.

Solution

Instead of requesting information from the server, make a reasonable guess.

There are times when it's better to provide a good guess than nothing at all. Typically, this pattern relates to dynamic situations where the browser periodically grabs new information, using *Periodic Refresh*. The aim is help the user spot general trends, which are often

more important than precise figures. It's a performance optimization because it allows you to give almost the same benefit as if the data was really arriving instantaneously, but without the bandwidth overhead. For this reason, it makes more sense when using *Periodic Refresh* than *HTTP Streaming*, because the latter doesn't incur as much overhead in sending frequent messages.

One type of Guesstimate is based on historical data. There are several ways the browser might have access to such data:

- The browser application can capture recent data by accumulating any significant observations into variables that last as long as the Ajax App is open.
- The browser application can capture long-term data in cookies, so it's available in subsequent sessions.
- The server can expose historical data for interrogation by the browser application.

Equipped with historical data, it's possible to extrapolate future events, albeit imprecisely. Imagine a collaborative environment where multiple users can drag-and-drop objects in a common space, something like the Magnetic Poetry Ajax App (<http://www.broken-notebook.com/magnetic/>). Using a *Periodic Refresh* of one second, users might see an erratic drag motion, with the object appearing to leap across space, then stay still for a second, then leap again. A Guesstimate would exploit the fact that the motion of the next second is probably in the same direction and speed as that of the previous section. Thus, the application can, for that second, animate the object *as if* it were being dragged in the same direction the whole time. Then, when the real position becomes apparent a second later, the object need not leap to that position, but a new estimate can be taken as to where the object's moving, and the object can instead move smoothly toward the predicted location. In other words, the object is always moving in the prediction of its current predicted location. Dragging motion is an example where users would likely favor a smooth flow at the expense of some accuracy, over an erratic display that is technically correct.

How about long-term historical data, stretching over weeks and months instead of seconds and minutes? Long-term data can also be used for a Guesstimate. Imagine showing weather on a world map. The technically correct approach would be not to show any weather initially, and then to gradually populate the map as weather data is received from the server. But the philosophy here would suggest relying on historical data for a first-cut map, at least for a few indicative icons. In the worst case, the Guesstimate could be based on the previous day's results. Or it might be based on a more sophisticated statistical model involving several data points.

Historical data is not the only basis for a Guesstimate. It's also conceivable the browser performs a crude emulation of business logic normally implemented server side. The server, for example, might take 10 seconds to perform a complex financial query. That's a problem for interactivity, where the user might like to rapidly tweak parameters. What if the browser could perform a simple approximation, perhaps based on a few assumptions and rule-of-thumb reasoning? Doing so might give the user a feel for the nature of the data, with the long server trip required only for detailed information.

There are a few gotchas with Guesstimate. For example, the Guesstimate might end up being an impossible result, like “-5 minutes remaining” or “12.8 users online”! If there’s a risk that your algorithm will lead to such situations, you probably want to create a mapping back to reality; for instance, truncate or set limits. Another gotcha is an impossible change, such as the number of all-time web site visitors suddenly dropping. The iTunes example discussed later in this chapter provides one mitigation technique: always underestimate, which will ensure that the value goes up upon correction. With Guesstimate, you also have the risk that you won’t get figures back from the server, leading to even greater deviation from reality than expected. At some point, you’ll probably need to give up and be explicit about the problem.

Decisions

How often will real data be fetched? How often will Guesstimates be made?

Most Guesstimates are made between fetches of real data. The point of the Guesstimate is to reduce the frequency of real data, so you need to decide on a realistic frequency. If precision is valuable, real data will need to be accessed quite frequently. If server and bandwidth resources are restricted, there will be fewer accesses and a greater value placed on the Guesstimate algorithm.

Also, how often will a new Guesstimate be calculated? Guesstimates tend to be fairly mathematical in nature, and too many of them will impact on application performance. On the other hand, too few Guesstimates will defeat the purpose.

How will the Guesstimate be consolidated with real data?

Each time new data arrives, the Guesstimate somehow needs to be brought into line with the new data. In the simplest case, the Guesstimate is just discarded, and the fresh data is adopted until a new Guesstimate is required.

Sometimes, a more subtle transition is warranted. Imagine a Guesstimate that occurs once every second, with real data arriving on the minute. The 59-second estimate might be well off the estimate by 1 minute. If a smooth transition is important, and you want to avoid a sudden jump to the real value, then you can estimate the real value at two minutes, and spend the next minute making Guesstimates in that direction.

The iTunes demo in the “Real-World Examples” section, later in this chapter, includes another little trick. The algorithm concedes a jump will indeed occur, but the Guesstimate is deliberately underestimated. Thus, when the real value arrives, the jump is almost guaranteed to be upward as the user would expect. Here, there is a deliberate effort to make the Guesstimate less accurate than it could be, with the payoff being more realistic consolidation with real data.

Will users be aware a Guesstimate is taking place?

It’s conceivable that users will notice some strange things happening with a Guesstimate. Perhaps they know what the real value should be, but the server shows something

completely different. Or perhaps they notice a sudden jump as the application switches from a Guesstimate to a fresh value from the server. These experiences can erode trust in the application, especially as users may be missing the point that the Guesstimate is for improved usability. Trust is critical for public web sites, where many alternatives are often present, and it would be especially unfortunate to lose trust due to a feature that's primarily motivated by user experience concerns.

For entertainment-style demos, Guesstimates are unlikely to cause much problem. But what about using a Guesstimate to populate a financial chart over time? The more important the data being estimated, and the less accurate the estimate, the more users need to be aware of what the system is doing. At the very least, consider a basic message or legal notice to that effect.

What support will the server provide?

In some cases, the server exposes information that the browser can use to make a Guesstimate. For example, historical information will allow the browser to extrapolate to the present. You need to consider what calculations are realistic for the browser to perform and ensure it will have access to the appropriate data. A *Web Service* exposing generic history details is not the only possibility. In some cases, it might be preferable for the server to provide a service related to the algorithm itself. In the Apple iTunes example shown later in the "Real-World Examples" section, recent real-world values are provided by the server, and the browser must analyze them to determine the rate per second. However, an alternative design would be for the server to calculate the rate per second, reducing the work performed by each browser.

Real-World Examples

Apple iTunes counter

As its iTunes Music Store neared its 500 millionth song download, Apple decorated its homepage (<http://apple.com>) with a rapid counter that appeared to show the number of downloads in real-time (Figure 13-6). The display made for an impressive testimony to iTunes' popularity and received plenty of attention. It only connected with the server once a minute, but courtesy of a Guesstimate algorithm described later in "Code Example: iTunes Counter," the display smoothly updated every 100 milliseconds.

Gmail Storage Space

The Gmail homepage (<http://gmail.com>) (Figure 13-7) shows a message like this to unregistered users:

2446.034075 megabytes (and counting) of free storage so you'll never need to delete another message.

But there's a twist: the storage capacity figure increases each second. Having just typed a couple of sentences, it's up to 2446.039313 megabytes. Gmail is providing a not-so-subtle message about its hosting credentials.

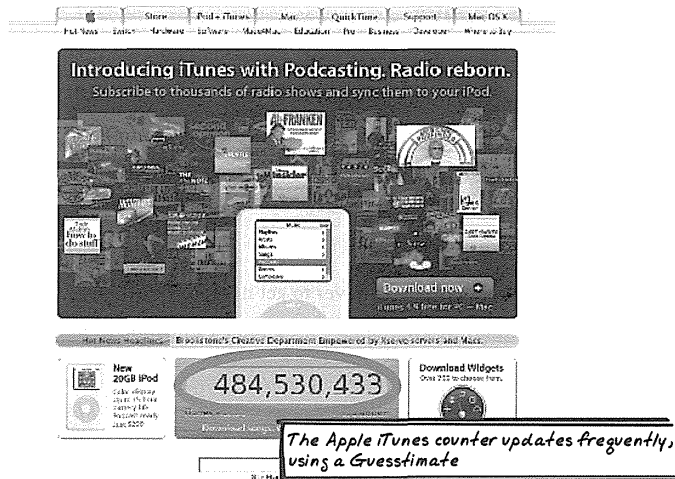


FIGURE 13-6. iTunes counter

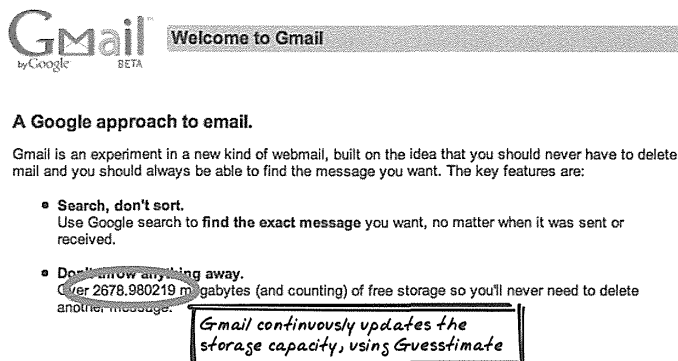


FIGURE 13-7. Gmail storage

Andrew Parker has provided some analysis of the homepage (<http://www.avparker.com/2005/07/10/itunes-gmail-dynamic-counters/>). The page is initially loaded with the storage capacity for the first day of the previous month and the current month (also the next month, though that's apparently not used). When the analysis occurred, 100 MB was added per month. Once you know that, you can calculate how many megabytes per second. So the algorithm determines how many seconds have passed since the current month began, and it can then infer how many megabytes have been added in that time. Add that to the amount at the start of the month, and you have the current storage capacity each second.

Code Example: iTunes Counter

The iTunes counter* relies on a server-based service that updates every five minutes. On each update, the service shows the current song tally and the tally five minutes prior. Knowing how many songs were sold in a five-minute period allows for a songs-per-second figure to be calculated. So the script knows the recent figure and, since it knows how many seconds have passed since then, it also has an estimate of how many songs were sold. The counter then shows the recent figure plus the estimate of songs sold since then.

There are two actions performed periodically:

- Once a minute, `doCountdown()` calls the server to get new song stats.
- Once every 100 milliseconds, `runCountdown()` uses a rate estimation to morph the counter display.

The key global variables are `rate` and `curCount`:

- `rate` is the number of songs purchased per millisecond, according to the stats in the XML that's downloaded each minute.
- `curCount` is the counter value.

So each time the server response comes in, `rate` is updated to reflect the most recent songs-per-millisecond figure. `curCount` is continuously incremented according to that figure, with the output shown on the page.

The stats change on the server every five minutes, though `doCountdown()` pulls down the recent stats once a minute to catch any changes a bit sooner:

```
//get most recent values from xml and process
ajaxRequest('http://www.apple.com/itunes/external_counter.xml',
            initializeProcessReqChange);
//on one minute loop
var refreshTimer = setTimeout(doCountdown,refresh);
```

It fetches is the counter XML:

```
<root>
  <count name="curCount" timestamp="Thu, 07 Jul 2005 14:16:00 GMT">
    484406324
  </count>
  <count name="preCount" timestamp="Thu, 07 Jul 2005 14:11:00 GMT">
    484402490
  </count>
</root>
```

`setCounters()` is the function that performs the *Guesstimate* calculation based on this information. It extracts the required parameters from XML into ordinary variables—for example:

* An expanded analysis of the iTunes counter can be found on my blog at <http://www.softwareas.com/ajaxian-guesstimate-on-download-counter>. Some of the comments and spacing has been changed for this analysis.

```
preCount = parseInt(req.responseXML.getElementsByTagName(
    'count')[1].childNodes[0].nodeValue);
```

When a change is detected, it recalculates the current rate as *(number of new songs) / (time elapsed)*. Note that no assumption is made about the five-minute duration; hence, the time elapsed (`dateDiff`) is always deduced from the XML.

```
//calculate difference in values
countDiff = initCount-preCount;
//calculate difference in time of values
dateDiff = parseInt(initDate.valueOf()-preDate.valueOf());
//calculate rate of increase
// i.e. ((songs downloaded in previous time)/time)*incr
rate = countDiff/dateDiff;
```

This is the most accurate rate estimate possible, but next, the script reduces it by 20 percent. Why would the developers want to deliberately underestimate the rate? Presumably because they have accepted that the rate will always be off, with each browser-server sync causing the counter to adjust itself in one direction or another, and they want to ensure that direction is always upwards. Making the estimate lower than expected just about guarantees the counter will increase—rather than decrease—on every synchronization point. True, there will still be a jump, and it will actually be bigger on average because of the adjustment, but it's better than having the counter occasionally drop in value, which would break the illusion.

```
rate = rate*0.8;
```

As well as the once-a-minute server call, there's the once-every-100-milliseconds counter repaint. `runCountdown()` handles this. With the rate variable re-Guesstimated once a minute, it's easy enough to determine the counter value each second. `incr` is the pause between redispays (in this case, 100 ms). So every 100 ms, it will simply calculate the new Guesstimated song quantity by adding the expected increment in that time. Note that the Gmail counter example just discussed calculates the total figure each time, whereas the present algorithm gradually increments it. The present algorithm is therefore a little more efficient, although more vulnerable to rounding errors.

```
//multiply rate by increment
addCount = rate*incr;
//add this number to counter
curCount += addCount;
```

And finally, the counter display is morphed to show the new Guesstimate. The show was all over when the tally reached 500 million songs, so the counter will never show more than 500 million.

```
c.innerHTML = (curCount<500000000) ?
    intComma(Math.floor(curCount)) : "500,000,000+";
```

Related Patterns

Periodic Refresh

A Guesstimate can often be used to compensate for gaps between *Periodic Refresh* (Chapter 10).

Predictive Fetch

Predictive Fetch (see earlier) is another performance optimization based on probabilistic assumptions. Predictive Fetch guesses what the user will do next, whereas Guesstimate involves guessing the current server state. In general, Guesstimate decreases the number of server calls, whereas Predictive Fetch actually increases the number of calls.

Fat Client

Guesstimates require some business and application logic to be calculated in the browser, a characteristic of *Fat Clients* (see later).

Metaphor

If you know what time it was when you started reading this pattern, you can Guesstimate the current time by adding an estimate of your reading duration.

Multi-Stage Download



Download, Incremental, Multi-Stage, Parallel

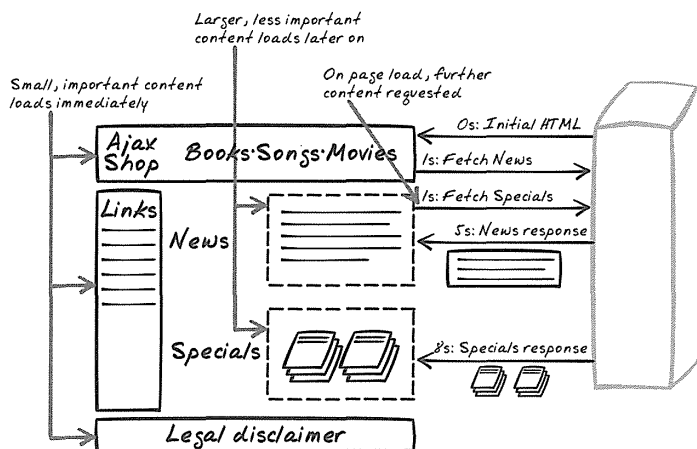


FIGURE 13-8. Multi-Stage Download

Developer Story

Dave's been informed an e-commerce homepage is turning many users away with its absurdly slow initial download. He refactors it so the initial download is nothing more than a banner and the key navigation links. News and featured items are still on the page, but they're pulled down separately, after the initial page has loaded.

Problem

How can you optimize downloading performance?

Forces

- Content can be time-consuming to load, especially when it contains rich images.
- Users rarely need to read all the content on the page.
- Users often have to click through several times to locate a particular item.
- Users sometimes browse around by exploring links between pages.

Solution

Break content download into multiple stages, so that faster and more important content will arrive first. Typically, the page is divided into placeholders (e.g., `div` elements) with the content for each placeholder downloaded independently. *XMLHttpRequest Calls* can be issued simultaneously or in a serial sequence, and the page will gradually be populated as the calls return.

The pattern applies when an application needs to download a lot of content from the server. Typically, this occurs on startup and also when a major context switch occurs. By breaking up the call, you can deliver faster and more important data earlier on. You avoid bottlenecks that occur when a single piece of content blocks everything else from returning.

The initial download of a page might be very small, containing just a skeleton along with navigation links and enough context to help the user decide if it's worth sticking around. That's the critical content for a fresh visitor, and there's no need to make him wait for the rest of the page to load. The browser can request the main page content as a separate call.

This pattern is in some respects an Ajaxian upgrade of the old technique of filling up a page gradually, where the server continues flushing the buffer and the user can see the page as it loads. Ajax makes the whole thing more powerful, because the latter downloads can attach to any existing part of the page.

One resource risk is having too many requests at once. Thus, consider a (purely speculative) Multiplex Call variant. Establish a browser/server protocol that lets the browser wrap multiple requests and the server wrap multiple responses. Then, issue only a single call with all required data inside. The immediate response might return a few results only, so the browser waits a bit, and then makes a further request for outstanding data. The whole process repeats until the browser has all the data.

Decisions

How will the page be divided into blocks?

The trickiest decision is how to divide the page into blocks, each of which will be downloaded individually. Since the blocks are downloaded in parallel, the main advice is to create small blocks of initial content to ensure the initial download is quick, and to group together content that's likely to be ready at the same time. Also, too many blocks will give the initial load an ugly appearance and may have the undesirable effect of causing already displayed elements to move around. For a fresh visitor, one example would be to create blocks as follows:

- A block of general information about the web site; e.g., name, summary, and a small icon.
- A navigation block showing links within the site.
- A block of general information about the current page, such as a heading and summary text. In a web app, this may be dynamic information based on what processing is being performed.
- One or more blocks of main page content.
- One or more blocks of auxiliary information; e.g., cross-site promotions, advertising, and legal notices.

Some of these may be combined too, as it's important to avoid too many blocks. So you might have a single request to retrieve both navigation and general information, which is then split and painted onto separate blocks.

Note that this doesn't have to apply to a homepage—if you are offering *Unique URLs*, visitors can jump into an Ajax App at different points, so the main page content might actually be quite “deep” into the application.

How will the page be structured?

Ideally, this pattern should not affect the page appearance, but will require some thinking about the underlying HTML. In particular:

- As new information loads, it's possible the browser application will automatically rearrange the page in order to accommodate new information. This can lead to the user clicking on the wrong thing, or typing into the wrong area, as items suddenly jump around the page. Ideally, information will not move around once it has been presented to the page. At least for images, it's worthwhile defining the width and height in advance, as CSS properties.
- CSS-based layout is likely to play a big part in any solution, as it provides the ability to exercise control over layout without knowing exact details.

What happens to the blocks while their content is being downloaded?

The entire DOM itself can be established on the initial load, so that divs and other structures are used as placeholders for incoming content. If that's the case, most of those elements can be left alone, but if the user is waiting for something specific that may take some time, it would be worth placing a *Progress Indicator* on the block where the output will eventually appear.

It's also possible to construct the DOM dynamically, so that new elements are added as new content arrives. This approach may be more work, but it may help to decouple the browser and the server. With the right framework in place, it means that the browser does not have to know exactly which items will be downloaded. The browser might ask the server to send all adverts, and the sender simply responds by sending down an XML file with three separate entries, which the browser then renders by adding three new divs.

Will the calls be issued simultaneously?

Let's say you're at a point where you suddenly need a whole lot of content. Should you issue several requests at once? Not necessarily. Keep in mind there are limits on how many outgoing requests the browser can handle at one time and consider what's best for the user. If there's a bunch of content that might not be used, consider deferring it for a bit with a JavaScript `setTimeout` call. That way, you can help ensure the user's bandwidth is dedicated to pulling down the important stuff first.

Real-World Examples

Kayak

Kayak (<http://kayak.com>) is a travel search engine. You tell it where you're going, and it then searches through many online travel sites to build up a set of options for you.

While waiting for a search query, the results page initially shows just the site banner, the search you entered, and a *Progress Indicator* (Figure 13-9). As the results for each external site arrives, Kayak adds the site name and whatever flights it found there. The flights aren't just appended, but kept in sorted order. You also have the option to finish the search by clicking an Enough Results button.

NetVibes

NetVibes (<http://netvibes.com>) is an Ajax portal that launches parallel *XMLHttpRequest Calls* on startup, one for each *Portlet*. Many portals follow this pattern.

TalkDigger

TalkDigger (<http://talkdigger.com>) is an Ajaxian meta-search for web feeds. Traditional meta-searches usually output results to the browser as they're received from individual engines, so the results page isn't well-ordered. With TalkDigger, each individual search engine gets its own div on the page. As the results come in, they're placed on the right spot.

kayak™ 3,979 kayakers online

Flights | Hotels | Cars | Deals | Buzz

London, United Kingdom to San Francisco, CA Mon 09 Jan 2006

Start search over

406 of 408 results shown

SINGAPORE AIRLINES

Enough Results

* Prices are for e-tickets and include all taxes & fees in USD.

Kayak keeps downloading flight routes and appending to results already downloaded.

Now adding results from:

- virgin-atlantic.com
- AA.com
- klm.com
- nwa.com
- aimewzealand.com
- delta.com
- united.com
- 1800CheapSeats.com
- continental.com
- airfrance.us
- alitalia.com
- flybmi.com
- flysn.com
- onetravel.com
- british-airways
- lufthansa.com
- aircanada.ca

Price	Airports	Airline	Depart
\$508	LHR > SFO SFO > LHR	Virgin Atlantic	11:00a 4:45p
\$516	LHR > SFO SFO > LHR	United	10:05a 6:10p
\$516	LHR > SFO SFO > LHR	United	10:05a 12:25p
\$516	LHR > SFO SFO > LHR	United	1:20p 6:10p

FIGURE 13-9. Kayak Progress Indicator

Code Example: AjaxPatterns Portal

The code example is a portal demo (<http://ajaxify.com/run/portal>) that downloads content in various stages (Figure 13-10):

1. The page banner and key links are downloaded immediately, as part of the initial HTML.
2. On page load, a place is set aside for side links, which are then requested and injected into the placeholder.
3. Likewise, the main content is also requested on page load.
4. Finally, an ad is requested after a short delay. This ensures the main content arrives first, so the ad doesn't affect the overall flow. (If you're feeling entrepreneurial, you can refactor it to reverse the order.)

The initial HTML contains the header and placeholders for the content that will be injected. There's an initial Loading message for each of these:

```
<h1>ajax patterns Portal demo</h1>

<a href="http://ajaxpatterns.org">Ajax Patterns Wiki</a> |
<a href="http://ajaxify.com">Ajax Demos</a> |
<a href="http://ajaxpatterns.org/Ajax_Examples">Ajax Examples</a>

<div class="spacer">&nbsp;</div>

<div id="leftLinks">
  <div id="sitelinks">Loading ...</div>
  <div class="spacer">&nbsp;</div>
  <div id="ad">Loading ...</div>
</div>
<div id="allPatterns">Loading ...</div>
```

ajax patterns Portal demo

Ajax Patterns Wiki | Ajax Demos | Ajax Examples

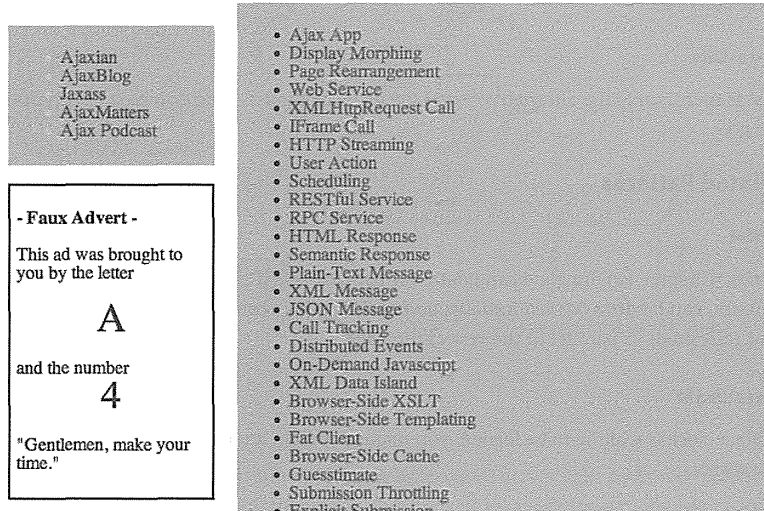


FIGURE 13-10. Multi-Stage Download Portal demo

The side links and main content are loaded immediately on page load, and sent to a call-back function that morphs the corresponding div:

```
window.onload = function() {  
    ajaxCaller.get("sideLinks.html", null,  
        onServerResponse, false, "sideLinks");  
    ajaxCaller.get("allPatterns.phtml", null,  
        onServerResponse, false, "allPatterns");  
    ...  
}  
  
function onServerResponse(html, headers, elementId) {  
    $(elementId).innerHTML = html;  
}
```

A few seconds later—probably after all the content has been loaded—the ad content is requested:

```
window.onload = function() {  
    ajaxCaller.get("sideLinks.html", null,  
        onServerResponse, false, "sideLinks");
```