downloaded from: lib.ommolketab.ir

Dave Crane, Bear Bibeault, and Jord Sonneveld

with Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker

Ajax in Practice

- Get going
- Get savvy
- 60 problems solved

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Ajax in Practice

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Ajax in Practice

DAVE CRANE
BEAR BIBEAULT
JORD SONNEVELD

WITH TED GODDARD, CHRIS GRAY,
RAM VENKATARAMAN AND JOE WALKER

MANNING
Greenwich
(74° w. long.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For online information and ordering of this and other Manning books, please go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830
Fax: (609) 877-8256
Email: orders@manning.com

©2007 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.

Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Copyeditor: Liz Welch
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-932394-99-0

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 11 10 09 08 07

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

brief contents

PART 1 FUNDAMENTALS OF AJAX 1

1 ■ Embracing Ajax 3
2 ■ How to talk Ajax 26
3 ■ Object-oriented JavaScript and Prototype 64
4 ■ Open source Ajax toolkits 117

PART 2 AJAX BEST PRACTICES 161

5 ■ Handling events 163
6 ■ Form validation and submission 202
7 ■ Content navigation 234
8 ■ Handling back, refresh, and undo 271
9 ■ Drag and drop 311
10 ■ Being user-friendly 336
11 ■ State management and caching 388
12 ■ Open web APIs and Ajax 415
13 ■ Mashing it up with Ajax 466

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

contents

preface *xiii*
acknowledgments *xv*
about this book *xviii*

PART 1 FUNDAMENTALS OF AJAX 1

1 *Embracing Ajax* *3*

1.1 Ajax as a disruptive technology *4*

Redefining the user's workflow 5 ▪ Redefining web application architecture 7

1.2 Ajax in ten minutes *9*

Introducing XMLHttpRequest 9 ▪ Instantiating XMLHttpRequest 10 ▪ Sending a request 11 Processing the response 13 ▪ Other XMLHttpRequest methods and properties 14

1.3 Making Ajax simple using frameworks *16*

Making requests with Prototype's Ajax.Request object 18 ▪ Simplifying Ajax responses 21

1.4 Summary *24*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

2 ***How to talk Ajax 26***

2.1 Generating server-side JavaScript *27*

Evaluating server-generated code 27 ▪ Utilizing good code-generation practices 30

2.2 Introducing JSON *34*

Generating JSON on the server 36 ▪ Round-tripping data using JSON 40

2.3 Using XML and XSLT with Ajax *44*

Parsing server-generated XML 44 ▪ Better XML handling with XSLT and XPath 50

2.4 Using Ajax with web services *56*

2.5 Summary *63*

3 ***Object-oriented JavaScript and Prototype 64***

3.1 Object-oriented JavaScript *66*

Object fundamentals 66 ▪ Functions are first class 68 Object constructors and methods 76 ▪ Writing a JavaScript class: a button 82

3.2 The Prototype library *97*

Generally useful functions and extensions 98 ▪ Array extensions 100 ▪ The Hash class 102 ▪ Binding context objects to functions 103 ▪ Object-oriented Prototype 105 Rewriting the Button class with Prototype 112

3.3 Summary *116*

4 ***Open source Ajax toolkits 117***

4.1 The Dojo toolkit *118*

Asynchronous requests with Dojo 119 ▪ Automatic form marshaling with Dojo 123

4.2 Prototype *125*

Asynchronous requests with Prototype 125 ▪ Automatic updating with Prototype 131 ▪ Periodic updating with Prototype 134

4.3 jQuery *136*

jQuery Basics 136 ▪ Asynchronous loading with jQuery 140 ▪ Fetching dynamic data with jQuery 145

4.4 DWR *150*

Direct Web Remoting with DWR 151

4.5 Summary *159*

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

PART 2 AJAX BEST PRACTICES 161

5 *Handling events* 163

5.1 Event-handling models *165*

Basic event-handling registration 165 ▪ Advanced event handling 169

5.2 The Event object and event propagation *172*

The Event object 172 ▪ Event propagation 173

5.3 Using Prototype for event handling *178*

The Prototype Event API 179

5.4 Event types *180*

Mouse events 181 ▪ Keyboard events 182 ▪ The change event 185 ▪ Page events 186

5.5 Putting events into practice *189*

Validating text fields on the server 190 ▪ Posting form elements without a page submit 195 ▪ Submitting only changed elements 198

5.6 Summary *201*

6 *Form validation and submission* 202

6.1 Client-side validation *203*

Validating on the client side 203 ▪ Instant validation 209 ▪ Cross-field validation 211

6.2 Posting data *218*

Anatomy of a POST 218 ▪ Posting data to a server 220 ▪ Posting form data to a server 223 Detecting form data changes 227

6.3 Summary *233*

7 *Content navigation* 234

7.1 Principles of website navigation *235*

Finding the needle in the haystack 235 ▪ Making a better needle-finder 237 ▪ Navigation and Ajax 238

7.2 Traditional web-based navigation *241*

A simple navigation menu 241 ▪ DHTML menus 243

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

7.3 Borrowing navigational aids from the desktop app *247*

The qooxdoo tab view 248 ▪ The qooxdoo toolbar and windows 250 ▪ The qooxdoo tree widget 254

7.4 Between the desktop and the Web *259*

The OpenRico Accordion control 259 ▪ Building an HTML-friendly tree control 263

7.5 Summary *270*

8 ***Handling back, refresh, and undo 271***

8.1 Removing access to the browser's navigation controls *272*

Removing the toolbars 272 ▪ Capturing keyboard shortcuts 274 ▪ Disabling the right-click context menu 275 ▪ Preventing users from navigating history or refreshing 276

8.2 Working with a browser's navigation controls *280*

Using the JavaScript history object 280 ▪ Hashes as bookmarks 281 ▪ Introducing the Really Simple History (RSH) framework 283 ▪ Using RSH to maintain state at the client level 284 ▪ Using RSH to maintain state at the server level 289

8.3 Handling undo operations *293*

When to provide undo capability 294 ▪ Implementing an undo stack 295 ▪ Extending the undo stack for more complex actions 300

8.4 Summary *309*

9 ***Drag and drop 311***

9.1 JavaScript drag-and-drop frameworks *313*

9.2 Drag and drop for Ajax *314*

Drag-and-drop Ajax shopping cart 314 ▪ Manipulating data in lists 321 ▪ The Ajax shopping cart using ICEfaces 326

9.3 Summary *335*

10 ***Being user-friendly 336***

10.1 Combating latency *338*

Countering latency with feedback 338 ▪ Showing progress 345 ▪ Timing out Ajax requests 351 Dealing with multiple clicks 355

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

10.2 Preventing and detecting entry errors *359*

Displaying proactive contextual help 359 ▪ Validating form entries 366

10.3 Maintaining focus and layering order *374*

Maintaining focus order 375 ▪ Managing stacking order 381

10.4 Summary *387*

11 ***State management and caching 388***

11.1 Maintaining client state *390*

11.2 Caching server data *392*

Exchanging Java class data 393 ▪ Prefetching 402

11.3 Persisting client state *406*

Storing and retrieving user state with JSON 406 ▪ Persisting JSON strings through AMASS 409

11.4 Summary *413*

12 ***Open web APIs and Ajax 415***

12.1 The Yahoo! Developer Network *416*

Yahoo! Maps 417 ▪ The cross-server proxy 421 ▪ Yahoo! Maps Geocoding 430 ▪ Yahoo! Traffic 436

12.2 The Google Search API *443*

Google search 443

12.3 Flickr photos *454*

Flickr identification 455 ▪ Flickr photos and thumbnails 459

12.4 But wait! As they say, there's more... *464*

Amazon services 464 ▪ eBay services 464 MapQuest 465 ▪ NOAA/National Weather Service 465 ▪ More, more, more... 465

12.5 Summary *465*

13 ***Mashing it up with Ajax 466***

13.1 Introducing the Trip-o-matic application *467*

Application purpose 467 ▪ Application overview and requirements 468

13.2 The Trip-o-matic data file *469*

What format should we use? 469 ▪ The trip data format 470 ▪ Setting up Flickr photo sets 471

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13.3 The TripomaticDigester class 473

The dependency check 473 ▪ The TripomaticDigester constructor 474 ▪ Digesting the trip data 475 Loading the points of interest 476 ▪ Collecting element text 477

13.4 The Tripomatic application class 479

The Tripomatic class and constructor 480 ▪ Creating the content elements 482 ▪ Filling in the trip data 484 Showing the map 487 ▪ Loading the thumbnails 488 Displaying the photos 491

13.5 The Trip-o-matic application page 492

The Trip-o-matic HTML document 492 ▪ Tripping along with style 494

13.6 Summary 496

index 499

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

preface

The Web has always been a hotbed of innovation, and, in its short history, we've seen many examples of an invention being repurposed and reused in ways far beyond the intentions of the original inventor. A network-based document retrieval protocol was subverted by the Common Gateway Interface into serving up dynamically-generated documents delivering data from a database back-end, allowing online access to one's data from anywhere in the world. HTTP headers were leveraged to provide the continuity of a user session on top of this stateless protocol, opening the door to stateful applications such as reservation systems and online commerce. Encrypted layers were built on top of the core protocol, to give confidence to the customers of these new online stores and users of business applications.

These were truly disruptive technologies, changing the way we use the Web forever. And yet today, technologies like server pages, sessions, and SSL are just everyday building bricks, baked into the fabric of every web developer's toolkit, to the point that we take them for granted. The pace of innovation is still relentless, though, with a new web framework appearing practically every week.

One of the biggest disruptions to the web development landscape in recent years has been Ajax. Through all the prior innovation, the basics of the web user interface—point and click, request, response, redraw—had not changed very much, until Microsoft quietly introduced the XMLHttpRequest (XHR)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

object with Internet Explorer 5 in 1999, using it to power their Outlook Web Access mail client to little fanfare.

The rest of the world suddenly sat up and took notice in 2005, when Google nailed their flag to the Ajax mast with their mail, maps, and suggest applications. Jesse James Garrett of Adaptive Path coined the term "Ajax," providing the first banner that we could all gather under to discuss exactly what this new thing was, and what we could do with it.

It seemed as if the technology was just waiting for a name, and once it had one, a flurry of activity ensued, with people trying to get into the Ajax spirit. However, Ajax introduced a new and different way of writing web applications. With new issues needing to be addressed, the last two years have seen yet another boom of innovations as the web development community figures out how to push this new and exciting envelope.

Along the way, the fundamentals of Ajax, like the XMLHttpRequest object, are going the way of the server page, the session, and SSL. The collective unconscious of the web development community has grokked the basic technology of Ajax, and is moving on to the broader issues that *use* of the technology raises.

It is in order to address these issues that we decided to write *Ajax in Practice*. With this book, our mission is to help accomplished (and not-quite-so-accomplished) web developers get on board with Ajax and successfully create their own Ajax-type applications. It can be regarded as a second-generation Ajax book: the first generation showed you what Ajax is; the second generation shows you what you can do with it and how to do it.

The book got its start when Steve Benfield was contacted by Manning to be the editor of a second-generation book about Ajax, as a follow-up to Dave Crane's popular *Ajax in Action* book. Later, Steve had to excuse himself as editor and Jord Sonneveld, Bear Bibeault, and Dave Crane teamed up to bring you this book in its completed form.

As you finish reading this preface, we have completed our mission and can sit back and share a few well-deserved drinks. We hope you enjoy reading this book as much as we have enjoyed writing it!

DAVE, BEAR, and JORD

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

acknowledgments

This section of a book always includes a surprisingly long list of names because it is indeed a collaborative effort of many different talents that results in the book that you are now holding. We have learned that firsthand! The authors do not work alone, although the long hours spent at the keyboard sometimes make it feel that way.

The publisher and editors at Manning Publications worked along with us, every step of the way, making sure the book was as good as it could be and we would like to thank them for their encouragement, insistence on quality, and attention to detail. There are many people who worked behind the scenes and we would like to acknowledge them here, along with publisher Marjan Bace and our editor Mike Stephens: Karen Tegtmayer, Howard Jones, Liz Welch, Dottie Marsico, Katie Tennant, Mary Piergies, Gabriel Dobrescu, Ron Tomich, and Olivia DiFeterici.

Our peer reviewers made many contributions, both large and small, to the manuscript during development, from catching errors in the code and typos in the text to suggestions on how to organize a chapter. The manuscript was reviewed a number of times and each pass resulted in a much better book. We would like to thank the following reviewers for taking time out of their busy schedules to read our chapters: Curt Christianson, Anil Radhakrishna, Robert W. Anderson, Srinivas Nallapati, Ernest Friedman-Hill, Jeff Cunningham, Christopher Haupt, Bas Vodde, Bill Fly, Ryan Lowe, Aleksey Nudelman, Lucas

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Carlson, Derek Lakin, Jonas Trindler, Eric Pascarello, Joel Webber, Jonathon Esterhazy, and Benjamin Gorlick.

Special thanks to Valentin Crettaz who was the technical editor of the book. He checked the code and reread certain chapters a number of times as we finalized them during production. His efforts are much appreciated.

Finally, thanks to Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker for their valuable contributions to the book on topics where they are the experts. We appreciate their collaboration with us on this project.

Dave Crane

I'd like to thank my colleagues Simon Warrick, Tim Wilson, Susannah Ellis, Simon Crossley, Rob Levine, and Miles Wilson at Historic Futures for their support for this project, and to Wendy, Nic, Graeme, and the team at Skillsmatter.com—and all my talented students—for helping to shape my thoughts on how this book should be written. Finally, and by no means least, I'd like to thank the rest of the Crane family—Chia, Ben, and Sophie—for putting up with me while I wrote two more programming books in parallel, my Mum and Dad, and my cats, for making good use of the hot air from the fan exhaust of my laptop during late night writing sessions.

Bear Bibeault

There are so many people to acknowledge and to thank. I want to thank all my friends and fellow staffers at javaranch.com, who offered encouragement when I expressed an interest in writing, and who include (but are not limited to): Ernest Friedman-Hill, Ben Souther, Max Habibi, Mark Herschberg, and Kathy Sierra.

Special thanks go to Paul Wheaton, owner of javaranch.com, for creating such a wonderful site and putting his trust in its staffers, and to Eric Pascarello for recommending me to Manning.

I want to thank my dogs Gizmo, Cozmo, and Little Bear, who provided companionship by lying on my feet as I penned these chapters and code. Cozmo gets special thanks for contributing random characters by swatting at the laptop keyboard as I typed. Thank goodness for editors.

And I want to thank my partner Jay, who put up with all the long hours it took to work on two books in parallel and with all the ranting about Internet Explorer and Word, and who offered nothing but encouragement for my efforts.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Jord Sonneveld

I would like to especially thank both of my co-authors, Dave and Bear, who shouldered much of the load in the later stages of development. This book would not have happened if it were not for their hard efforts.

To my parents and grandparents, thank you for buying me my first computer, and for all of your support while I was busy with this book.

Finally, I'd like to thank my cats for short-circuiting my UPS, and Mallory, my awesome lady friend who digs UNIX.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

about this book

Ajax has taken the web development community by storm, giving web developers the potential to create rich user-centered Internet applications. But Ajax also adds a new level of complexity and sophistication to those applications. *Ajax in Practice* tackles Ajax head-on, providing countless hands-on techniques and tons of reusable code to address the specific issues developers face when building Ajax-driven solutions.

After a brief overview of Ajax, this book takes the reader through dozens of working examples, presented in an easy-to-use solution-focused format. Readers will learn how to implement rich user interfaces, including hands-on strategies for drag and drop, effective navigation, event handling, form entry validation, state management, choosing Ajax libraries, interfacing to open web APIs, and more!

Unlike the traditional cookbook approach, *Ajax in Practice* provides a thorough discussion of each technique presented and shows how the individual components can be connected to create powerful solutions. A fun "mashup" chapter concludes the book. Throughout the book, the examples chosen are interesting, entertaining, and above all, practical.

With this book you will

- Go beyond what Ajax is, and learn how to put Ajax to *work*.
- Master *numerous* techniques for user interface design and site navigation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- Work hands-on with professional-grade reusable Ajax code designed to solve *real* problems.

Audience

This book is aimed at web developers who want their applications to be best-in-class examples of rich user interfaces, leveraging Ajax technology to achieve this goal.

While novices to Ajax will find the first two chapters helpful in getting kick-started into the world of asynchronous requests, this book is primarily aimed at developers with at least a basic background in developing web applications and in the rudimentary use of JavaScript to effect client-side activity.

In the new world of rich client-side user interfaces, the amount of client-side code has greatly increased and it is important to treat this code with the same level of respect due its server-side counterpart. Advanced JavaScript techniques that help to organize this client-side code and to use Ajax effectively are presented in this book.

If you are a web developer interested in expanding your coding skills not only with new technologies, but also with techniques and patterns that make the best use of that technology, we think that you will find that this book addresses those needs.

Whether you are a seasoned client-side developer, or one that is just starting out creating rich user interfaces to your web applications, we hope this book will have something for you.

Roadmap

We've divided this book into two parts. Part 1, "Fundamentals of Ajax," includes four introductory chapters that make sure that you've got the know-how under your belt that you'll need to make best use of the second part of the book. The chapters in part 2, "Ajax Best Practices," present various practical topics in client-side programming, with an emphasis on using Ajax directly, or on practices and principles that work well in Ajax-enabled applications.

Chapter 1 dives head first into what makes Ajax different from other technologies and why there's so much to be written (and learned) about it. It presents a crash course in using Ajax across the various browsers and how to deal with the responses it generates. Finally it closes with a brief look at how use of the Prototype library makes the whole process more streamlined.

In chapter 2, we examine the various categories of Ajax communication including JSON, XML and XSLT. We'll also investigate the use of Ajax with SOAP web services.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 3 introduces the concept of using object-oriented JavaScript to take control of the increasing amount of client-side code that the typical Ajax application contains. Key concepts such as the object construction, functions as first class objects, functions as class methods, function contexts, as well as closures are explained and put into perspective in relation to object-oriented techniques. Use of the Prototype library to help easily define JavaScript classes closes out this chapter.

Chapter 4 continues our investigation of Ajax-enabled JavaScript libraries with a closer look at Prototype, as well as the Dojo Toolkit, jQuery, and DWR libraries. While it would be impossible to cover the complete feature set of all these offerings, each is examined with particular attention to what they bring to the Ajax party. We'll see each of these libraries put into practice in the copious code examples in the remaining chapters.

The world of event handling is examined in chapter 5. The various event models are investigated with particular emphasis on cross-browser issues, along with the use of the Prototype library to ease those cross-browser pains. The most commonly used event types are discussed in relation to how they fit into Ajax applications.

Chapter 6 dives into the details of data entry validation of form data and how it ties into the event handling lessons of chapter 5. Both the Prototype and jQuery libraries are used to great advantage in the examples of this chapter, which include demonstrating how to hijack form submissions that would usually initiate a full-page refresh, and redirect them to less-intrusive Ajax requests.

In chapter 7, the subject of content navigation is addressed. We'll examine the creation of simple menus, and then progress to more complicated navigational aids such as tree views, accordion controls, tab views, and toolbars. The aid of libraries such as OpenRico and qooxdoo is enlisted by the code in this chapter.

Chapter 8 focuses on the mine field of problems created when users use back and refresh. We'll look at the problem both from the point of view of removing such abilities from the user, as well as working with such actions. This chapter also discusses adding a handy undo facility to applications.

Drag-and-drop operations are the topic of chapter 9. We'll examine the mechanics of drag and drop sequences, and discuss support for drag and drop in JavaScript libraries. We explore the use of Scriptaculous for manipulating lists, and develop a simple shopping cart implementation using Scriptaculous and ICEfaces.

In chapter 10, we discuss usability considerations and look at how Ajax can help us solve, or at least alleviate, latency issues. Reducing user frustration by providing server-assisted pro-active help is examined, and another look at validating

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

form data is taken. Dealing with tab and stacking order in the new arena of rich user interfaces is also addressed.

Chapter 11 covers state management. We'll explore how to maintain client state, cache data, prefetch data, and how to persist the client state. We also discuss using the AMASS library to persist large amounts of data.

In chapter 12 we delve into the exciting world of open APIs on the web. We learn how to avoid the dreaded "Ajax security sandbox" in order to make Ajax requests to remote servers. We then use that knowledge to make use of open APIs such as Yahoo! Maps, Geocoding and Traffic, the Google search engine, and Flickr photo services.

Chapter 13 culminates the book with a full "mashup" application that employs the open APIs we investigated in chapter 12, as well as the skills and techniques gathered throughout the book, to create a complete and working mashup application.

Code conventions

All source code in listings or in text is in a fixed-width font `like this` to separate it from ordinary text. Method and function names, properties, XML elements, and attributes in text are presented using this same font.

In many cases, the original source code has been reformatted: we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases even this was not enough, and listings include line-continuation markers. Additionally, many comments have been removed from the listings.

Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

Code downloads

Source code for all of the working examples in this book is available for download from http://www.manning.com/crane2 or http://www.manning.com/Ajaxin-Practice.

Author Online

Purchase of *Ajax in Practice* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

forum and subscribe to it, point your web browser to http://www.manning.com/crane2 or http://www.manning.com/AjaxinPractice. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It is not a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the cover illustration

The figure on the cover of *Ajax in Practice* is a "Sultana," a female member of a sultan's family; both his wife and his mother could be addressed by that name. The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book's table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book...two hundred years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the "Garage" on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor did not have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person's trust in one of us. It recalls something that might have happened a long time ago.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The pictures from the Ottoman collection, like the other illustrations that appear on our covers, bring to life the richness and variety of dress customs of two centuries ago. They recall the sense of isolation and distance of that period—and of every other historic period except our own hyperkinetic present.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago—brought back to life by the pictures from this collection.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Part 1

Fundamentals of Ajax

This book is intended as a head-on rush into the world of Ajax web applications, with particular emphasis on providing heaps of reusable, hands-on examples that illustrate *practical* techniques you can employ in your own applications. So that you are ready for that exciting journey, part 1 serves as an intensive preparation for the chapters that follow in part 2.

Chapter 1 discusses how Ajax differs from technologies that you might be accustomed to and sets up the expectations for the rest of the book. We discuss how you can use Ajax in supporting browsers and how asynchronous responses are dealt with in JavaScript code. We also take a brief look at Prototype, a popular JavaScript library that we'll be seeing again and again throughout the book.

Chapter 2 examines the types of response formats that Ajax requests can generate: plain text, HTML, JSON (JavaScript Object Notation), XML, or even SOAP documents.

In chapter 3, we investigate the advanced JavaScript techniques that every serious Ajax developer needs to have under their belts. We look at JavaScript objects and functions, and explain how to use them to create your own JavaScript classes in order to use object-oriented techniques to grab control of the ever-growing amount of client-side code that Ajax requires. You'll learn how JavaScript functions are much more complex and diverse a concept than you might have imagined.

Chapter 4 surveys a handful of JavaScript libraries that offer Ajax support. We explore the venerable Prototype library in greater detail, the versatile Dojo

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

toolkit, and jQuery, an exciting (relative) newcomer to the Ajax arena. The chapter concludes with a look at how DWR uses Ajax to provide an approximation of RPC (remote procedure calling) using Ajax as a transport mechanism.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

1 Embracing Ajax

This chapter covers

- What makes Ajax different
- Basic usage of XMLHttpRequest
- Simplifying Ajax using libraries

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Ajax has been growing up fast in the last year or so. At the time of this writing, Ajax is officially one and a half years old, although a lot of the underlying techniques have existed for several years longer, without a unifying name to describe them. The story has been related many times already, from the origin of an ActiveX control called XMLHttpRequest in Microsoft's Web Outlook, to Jesse James Garrett's coining of the term *Ajax* in February 2005, and the sudden explosion of interest in these techniques centered around Google's Suggest, GMail, and Maps applications.

As with any kid growing up in the modern world, it's been a struggle at times, and the Ajax that we're seeing today looks a lot different from the being we met a year and a half ago. The technology has matured, our vocabulary for discussing the technology has matured, and the tools available to do the job have matured too. We'll expand on this a little in section 1.4, and we'll take an in-depth look at the new breed of frameworks and libraries that are making Ajax easier to use in chapter 4.

The biggest change that has taken place as Ajax matures, though, is that our understanding of what we can do with Ajax has expanded. Developers are asking themselves new sets of questions, going beyond the basics of, How do I do it? to deeper and broader issues, such as, How do I manage my asynchronous communications?, How does Ajax affect my application architecture?, and even, What does Ajax mean for my business model?

Collectively, the development community has embraced Ajax and, as with the best inventions, used it in new and interesting ways. Google demonstrated that, using Ajax, "solved problems" like online maps and webmail still had plenty of room for radical innovation. The recent interest in "mash-ups" (the mixing of content from more than one website in a single page) has a natural affinity with Ajax, too.

Along the way, we've amassed practical experience in using Ajax in real-world applications and settings. Our purpose in writing this book is to capture some of this practical experience with Ajax, and go beyond the basic proof-of-concept code to look at what does—and doesn't—work in the real world. As such, we intend to focus on the deeper, broader questions that are relevant to Ajax today.

1.1 Ajax as a disruptive technology

Ajax is a disruptive technology. That is, it has appeared and is disrupting the normal way online applications are built and delivered and is changing how people perceive web applications and what can be done with them.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In the narrowest sense, Ajax is simply the business of making asynchronous requests to the server. By *asynchronous*, we mean that the request is taking place in the background, out of sight of the user interface. When we're coding an Ajax application, we're only spending a small fraction of our time making asynchronous calls to the server and processing the response. The rest of the time, we're using established technologies like Cascading Style Sheets (CSS), the Document Object Model (DOM), and the browser event model. In short, we're using the set of technologies known as *Dynamic HTML*, a set of technologies that were practically dead in the water two years ago, relegated to rendering fancy navigation menus and those pop-up ad windows that we all love. Adding asynchronous HTTP capabilities into the mix has revitalized these technologies, giving them a new reason to be used. So why has this little nugget of Ajax made such a big impact? The answer is surprisingly simple, as we'll see in the next chapter.

1.1.1 Redefining the user's workflow

The key to understanding the impact that Ajax has had on web development lies in the user workflow. By *workflow*, we mean the way in which the user interacts with the application and, in the broader sense, how they experience the application. We commonly talk about web apps in terms of a division of work between the browser and the server, but these are simply enablers for the important work that is going on in our users' heads. A good app makes the user productive by supporting their working patterns, whereas an application that dictates the user's work pattern based on its own technical limitations reduces productivity. Figure 1.1 shows the workflow of a pre-Ajax classic web application.

This workflow follows a work-wait pattern. That is, at any point in time, the browser-side application is either presenting information to the user or waiting for the server to return a response. From the user's point of view, the experience

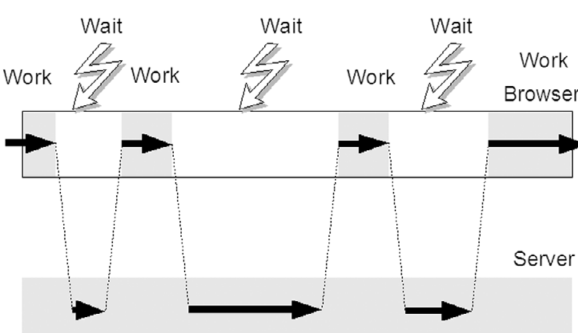

Figure 1.1
Work-wait pattern of user interaction in a classic web application

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

is very punctuated. During the wait periods, the user is unable to interact with the web app, beyond possibly being able to read some of the content on the page that is just about to be replaced by the server response.

From a usability perspective, this is extremely problematic. Each wait period is an interruption to the user's train of thought. And yet the wait periods must be frequent. The majority of web applications will require frequent contact with the server. This model is clearly unsuitable for any kind of activity that entails complex problem solving—which is a pity, as browser-based applications have a lot of advantages. DHTML provides all the ingredients for a pleasant, responsive user interface, and perhaps most notably, web applications are extremely easy to deploy and maintain, as no installation on the client machine is required.

In figure 1.2, we show how Ajax changes the workflow for the user. Here, the application is still making the same requests to the server, but is doing so using Ajax. This allows the user interface to remain active during the times when the server is busy, and therefore removes the continual disruptions to the user's concentration.

From a business perspective, the importance of this cannot be understated. Ajax has opened up a large new market to browser-based line-of-business apps, disrupting not only web development, but the world of thick clients and desktop apps in the process. Within the enterprise, adoption of Ajax-based solutions is considerable. On the public Internet, several heavyweight Ajax-based office suites have been developed in the last year, and web-based operating systems are under development. Although none of these have yet gone mainstream, progress has been considerable.

So, Ajax has had a significant, and disruptive, effect on the application marketplace, which presents challenges and opportunities for us as developers. Looking within our own domain of expertise, however, Ajax can be considered disruptive in other ways, as we'll discuss in the next section.

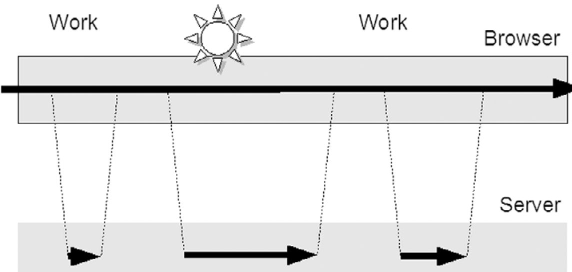

Figure 1.2
Work-work pattern of interaction in an Ajax application

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

1.1.2 Redefining web application architecture

Web application architecture has always been an interesting field. Because of the challenge of maintaining an adequate user workflow in the face of the work-wait nature of the Web, there has been a continual stream of innovation in the way in which web applications are organized on the server. Nonetheless, certain conventions have been established, such as the division of responsibility between the presentation tier and a business tier, consisting of a persistable domain model. Figure 1.3 illustrates this design, as well as the ways in which Ajax is affecting it.

In the pre-Ajax architecture, pictured on the left, all the action is taking place on the server, with the browser acting as a dumb terminal, accepting predigested HTML content.

The middle column illustrates the impact of introducing some relatively simple Ajax into the application. Let's say that the server still controls all aspects of the workflow, but hyperlinks and forms now request fragments of HTML content that are used to update parts of the screen, rather than perform a full refresh. Server responses are fielded by JavaScript code, which reads the response and rearranges the DOM accordingly. The presentation tier on the browser has started to get thicker, as we add the JavaScript to route the content received from the server. We will also often see the presentation tier on the server starting to get smaller, as we're generating simpler, more focused responses, rather than assembling entire pages every time.

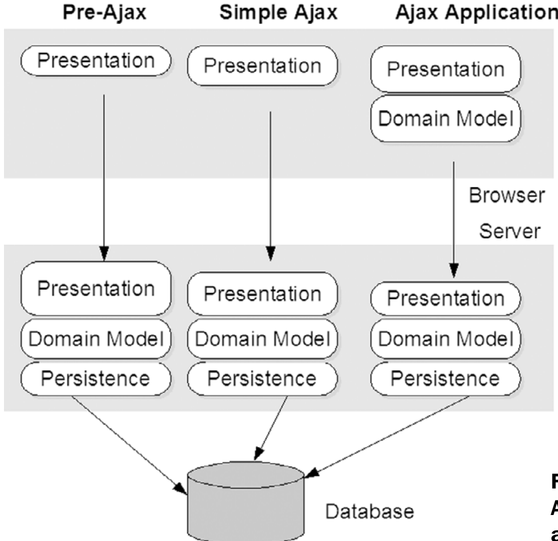

Figure 1.3
Architecture of an n-tier web application, and the impact of Ajax on the design

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

A well-factored classic web application will tend to generate its responses in a modular fashion anyway, so introducing some Ajax functionality is not going to entail a complete rewrite. However, the pattern of server requests and responses over time is likely to change, and we've introduced a new presentation tier on the browser, written in JavaScript. We may not be disrupting the development team with this approach to Ajax, but we are making them think again, and picking up some new skills along the way.

As we get deeper into Ajax—and move toward the new breed of line-of-business web apps that Ajax has enabled—the scope for changing the architectural tiers increases. On the right-hand side of figure 1.3, we've depicted an extreme case of an Ajax-based application, in which the JavaScript code in the browser is sufficiently complex to be divided into tiers itself. In this case, the client-side presentation tier has control over the users' workflow. The client-side code also maintains a partial model of the major domain entities, and the JavaScript presentation tier will tend to communicate to these rather than directly to the server.

On the server side, we can see that the presentation tier is much reduced. Its main responsibilities would be to provide a coarser-grained façade on top of the domain model, which defines the main use cases for the application. It may also control the marshaling and unmarshaling of data across the HTTP interface. Flow control and visual presentation of content have been largely delegated to the client-side JavaScript tiers.

Not every Ajax application will follow this approach to its full extreme, nor would it be appropriate to do so. We said at the outset that the architecture of web applications is not a well-defined, solved problem, and that plenty of room still exists for innovation. Ajax disrupts the web architecture landscape by moving the innovation in new directions, not by providing a single solution. Think of figure 1.3 as presenting three points along a spectrum, with Ajax-enhanced legacy applications tending to sit near the middle, and line-of-business Ajax apps toward the right-hand side.

We've set the scene for this book now, and we'll return to these concerns throughout our examples. Now, let's jump into our first coding exercise, with a look at how to make an Ajax request. It's worth doing this once, just to highlight a few issues, and the ways in which key technologies such as JavaScript and HTTP fit together. After this, we'll pick up speed as we wrap the low-level functionality up in libraries, and move on to higher-level concerns. First, though, let's look at the XMLHttpRequest, and see what it can do.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

1.2 Ajax in ten minutes

If you're already familiar with how Ajax works, then you can safely skip the rest of this chapter. If you're not, or you're up for a refresher, this text covers the core pieces of functionality that allow Ajax applications to be built. Ajax is not a specific product, nor is there a specific set of Ajax functions in the browser. Instead, as you'll see, Ajax is the use of a specific JavaScript object called *XMLHttpRequest* combined with JavaScript events and dynamic HTML (DHTML) (also called *DOM manipulation*). In this section, we'll take the XMLHttpRequest object out for a walk, and come to grips with its basic capabilities.

1.2.1 Introducing XMLHttpRequest

When we write classic web applications, we use the HTTP protocol to communicate between the browser and the server. The primary means of user interaction are hyperlinks and HTML forms, both of which trigger HTTP requests in the browser. A limitation of both of these is that they automatically populate the current page, or a frame in the current page, with the response. That is, they are designed for retrieving content across the Web.

As we start to work with more complex client applications, we may need to retrieve data rather than content, or retrieve finer-grained content to insert into the current page. The XMLHttpRequest object (which we'll abbreviate to XHR from here on) was developed as a solution to this problem, allowing greater programmatic control over HTTP requests.

As we discussed in the previous section, the XHR object allows us to make HTTP requests to the server and to receive the response programmatically, rather than the browser automatically rendering the response as a new page. From the perspective of the client-side code, then, there are several things that we need to do in order to achieve this, as summarized in figure 1.4.

The first thing that we need to do is to create an XHR object ❶. We then provide it with the information that it needs to make the request ❷. Finally, we handle the response when it comes back in ❸. In between sending the request and receiving the response, there is work to be done on the server too, of course, and some more code for us to write, in PHP, Java, a .NET language, or whatever our current environment dictates. We're interested here primarily in the client-side code, though, as the server-side mechanics of handling a simple Ajax request are not very different from pre-Ajax web programming. We'll present server-side code later in the book, for the more involved examples, but

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

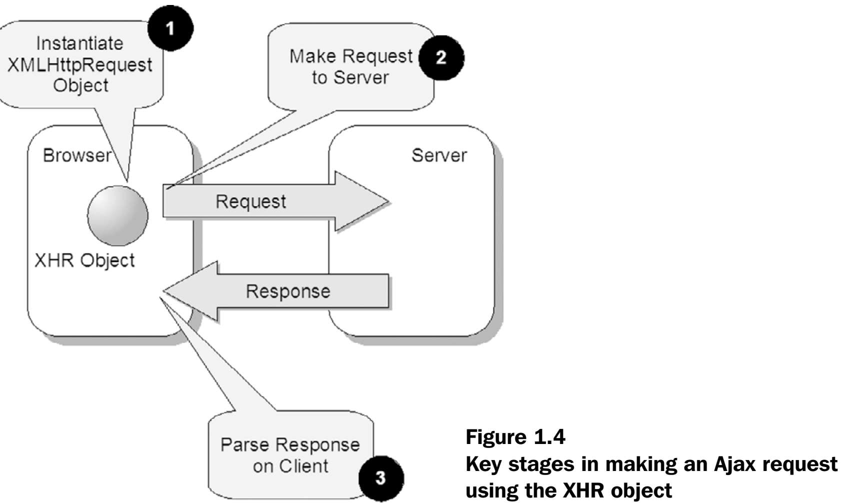

Figure 1.4
Key stages in making an Ajax request using the XHR object

for now, we just want to figure out how the client works. We'll refer back to figure 1.4 as we work through the steps.

The first thing that we need to do is to get ahold of an XHR object.

1.2.2 *Instantiating XMLHttpRequest*

The XHR object is built into the four major modern browser families: Internet Explorer, Firefox/Mozilla/Netscape, Safari, and Opera. To use the object, you'll create an instance of the XHR object, give it some parameters to set up the request you want to send, tell it to send the request, and then process the result. Listing 1.1 shows a cross-browser example of the first step, namely, instantiating the XHR object.

Listing 1.1 Instantiating an XHR object

```
var xhr;
if (document.XMLHttpRequest) {          <-- Detects XHR object
   xhr = new XMLHttpRequest();          <-- Creates native object
} else {
  xhr = new ActiveXObject("Microsoft.XMLHTTP");     <-- Creates ActiveX control
} else {
  alert("cannot use Ajax");
}
```

The reason for the complexity of this code is cross-browser incompatibility (which should come as no surprise to seasoned web developers). Internet Explorer (prior

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

to version 7, at least) does not have a native XHR object. Rather, it implements XHR as an ActiveX object. Because of this, if the user has "Safe ActiveX" scripting turned off, they will not be able to run an Ajax-enabled application. (Before casting blame on Microsoft for this implementation, remember that they invented the XHR object in the late '90s and implemented it as an XML parsing module that started shipping with IE. Only recently have other major browsers added support for XHR.)

We also need to check for older browsers that don't support any kind of XHR object, and issue some sort of message to them, stating that the app won't run on this browser. Depending on the browser in which the code is being run, we will follow one pathway or another through the `if()` statements, and, at the end of it, have a reference to an XHR object. It might be a native object or an ActiveX control, but as long as we have an XHR of some kind, we can start to use it. Fortunately, whatever kind of XHR we have, the methods and properties of the object are pretty much identical from here on.

1.2.3 Sending a request

Let's return to figure 1.4 briefly. The XHR object is now instantiated, so we're on to the second stage: sending the request. Before we start examining how the XHR object does this, let's look at the basic information needed to set up a call to the server. We will need

- The URL of the server resource
- The HTTP Request type, usually a GET or a POST
- Parameters needed by the server resource
- A JavaScript function to interpret the results returned from the server

OK, let's start checking these items off the list. The first two items, and possibly the third, are passed when calling the `open()` method. `open()` initializes a connection to a URL. The method is overloaded and has three forms:

```
open(http_method, url)
open(http_method, url, asynchronous)
open(http_method, url, asynchronous, userid, password)
```

The method is almost always a GET or POST, but could be any valid HTTP method such as PUT, DELETE, HEAD, and so forth, that is supported by the server. If `asynchronous` is true, then the request will run in the background, thus allowing the user to perform other work while the XHR request is being processed. If it is false, then the request will be synchronous and the user will be

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

blocked until the request is finished, much the same way as when working with traditional work-wait web applications.

This third argument reflects the legacy of XHR as a general-purpose ActiveX control. Within some applications, it may make sense to make synchronous requests, but the JavaScript interpreter is essentially single-threaded, and making a synchronous request will block all user interaction with the browser until the response has returned. In an Ajax app, always make your requests asynchronous.

`userid` and `password` are used to connect to servers that require them. This is only valid for HTTP authentication (as opposed to NT domain-based authentication, for example), which sends passwords as plain text, and should be treated with caution unless operating over a secure socket via HTTPS.

So, to open a connection to a URL, we might write

```
xhr.open('GET', 'servlets/ajax/getItem?id=321', true);
```

Because we're using the HTTP GET method, we're passing parameters to the server in the URL as a query string. If we were using POST, we'd pass them in the request body, which we'll look at in a minute.

The second stage to priming the XHR object is to assign a callback handler function to receive the response. For now, we won't worry about what the function does, but simply assign it. For example:

```
xhr.onreadystatechange = parseResponse;
```

Note that we pass a reference to the function object. We don't call the function at this point—there are no parentheses after the function—but inform the XHR that this is the function to call when the response comes back. This callback assignment is identical to setting UI event handlers, such as `onclick` and `onmouseover` on DOM elements.

The third stage is to call the `send()` method. `send()` executes the server call and can be used to send additional data not specified in the URL. `send()` takes a single argument, the additional data to be sent in the request body. Normally, only POST requests have a body, so for GET requests, we just pass an empty string:

```
xhr.send('');
```

That's it! The request is now on its way to the server, and there's nothing more for the client code to do until the response comes back. We'll address that issue in the next section.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

1.2.4 Processing the response

We've fired the request at the server, and we can assume for now that the server will do its job and return a response to us. Referring again to figure 1.4, the next thing that we need to do is to receive the response when it comes in and unpack it. In the previous section, we already made preparations for this moment, by assigning a callback handler function to our XHR object. In this section, we'll see what happens when that callback is invoked.

You might be forgiven for thinking that the XHR would simply inform you when the response had arrived, but instead, it informs you at several points in the lifecycle of the response. In a minority of cases, this is extremely useful information to have, but normally, it's a distraction.

We assigned a callback handler called `onreadystatechange` in the previous section. This function will be called at least once for every ready state that the XHR object undergoes. In your `onreadystatechange` function, you will need to manually check the `readyState` property to determine where the request currently stands in its lifecycle and whether you can process the final result. `readyState` will always be one of these predefined values:

Value	State	Description
0	Uninitialized	`open()` has not been called.
1	Loading	`open()` has been executed.
2	Loaded	`send()` has been executed.
3	Interactive	The server has returned a chunk of data.
4	Complete	The request is complete and the server is finished sending data.

You will almost always only check for `readyState == 4`, meaning that the request is finished. So a typical callback function might look like this:

```
xhr.onreadystatechange = function(){
  var ready = xhr.readyState;
  if (ready == 4){
    parseCompletedResponse(xhr);
  }
};
```

That is, we check whether the `readyState` property value is `4`, which indicates that the response has arrived in its entirety and can be parsed. If so, we hand over to a parsing function. When we parse the response, the first thing we'll want to know is whether the request was handled successfully.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The status property contains the HTTP status of the request. A valid HTTP GET or POST normally returns 200 if the requested URL was processed correctly. A 404 is returned if the URL does not exist. Typically, any result code between 200 and 299 represents success; any other code indicates failure or further action by the browser. By combining readyState and status, you can determine whether the request has finished successfully. We might modify our function to something like this:

```
xhr.onreadystatechange = function(){
  var ready = xhr.readyState;
  if (ready == 4) {
    var status = xhr.status;
    if (status >= 200 && status < 300) {
      parseCompletedResponse(xhr);
    } else {
      parseErroredResponse(xhr);
    }
  }
};
```

Let's assume that our response has come in fine. The response is provided in two properties: responseText and responseXML. responseText presents the response as a plain string. responseXML presents the response as a parsed XML document. We'll look at both of these in more detail in the examples in this and the next chapter.

We've now followed through the full lifecycle of a simple Ajax request and response. We've had to handle a lot of plumbing along the way, in getting ahold of the object, in assembling the request, and in parsing the response. The good news is that, having demonstrated these low-level details once, we aren't going to return to them again in this book, as there are several good frameworks and libraries out there that will do the grunt work for us.

However, knowing how XHR works is useful, because it will help us to understand what the Ajax libraries wrappers can and can't do. To this end, we'll present a few additional methods and properties of the XHR object before moving on to higher things.

1.2.5 Other XMLHttpRequest methods and properties

There are other, lesser-used, XHR methods. In simple cases, you won't need to know about these, but they can be very useful for specific tasks.

abort()

abort() aborts the current request, if possible. This is a client side-only abort—if the send() method has already been called, the server will have received the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

HTTP request and will process the result. However, the browser will ignore the result and stop processing.

setRequestHeader(header, value)

This function sets a header value for the HTTP request. It is most commonly used for setting the content type of the request body. Any valid HTTP header value can be used. You might use this function for a number of purposes, for example, to set the request MIME type to `x-www-form-urlencoded` so you can emulate posting an HTML form:

```
xhr.setRequestHeader(
  'Content-type',
  'application/x-www-form-urlencoded'
);
```

With Ajax, we're not limited to POSTing key-value pairs to the server. We might send an XML payload, in which case, we should tell the server that we're sending XML:

```
xhr.setRequestHeader(
  'Content-type',
  'application/xml; charset=UTF-8'
);
xhr.send("<data source='ajax in practice'>hello world</data>");
```

We'll discuss using XML with Ajax in greater detail in chapter 2. There are also some methods we can make use of when handling the response.

getResponseHeader(header)/getAllResponseHeaders()

An HTTP response typically contains many headers, each of which is a key-value pair. The XHR object can list all header names using `getAllResponseHeaders()`, and it can read a header value using `getResponseHeader()`, which takes a header name as argument. For example, to determine if the server is a Microsoft IIS server, you could use the following:

```
if (xhr.getResponseHeader("Server")
  .indexOf("Microsoft-IIS") != -1 ) {
    alert("The server is a Microsoft IIS server.");
}
```

Now that we've worked through the use of the XHR object in detail, we can turn our attention to more interesting questions. In the remainder of this chapter, we'll introduce the Prototype library's helper objects for making Ajax easier, and start working through a series of examples that will explore what we can do with our newfound power to fire asynchronous requests at the server.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

1.3 Making Ajax simple using frameworks

In the previous section we saw the basics of how to create an Ajax request using the XHR object. Now that we have the ability to fetch data from the server without initiating a full-page refresh, we find ourselves wondering what we're going to say to the server, and what sort of a response it might offer.

The ground rules laid down by the HTTP protocol are fairly loose. Any communication must be started by the client's request, and completed by the server's response, and both halves of the communication must be text-based. Beyond that, though, more or less anything goes. In this chapter we will look at the different forms of data that can be passed using Ajax, and begin to consider how we'll use Ajax to structure the browser/server communication in our application as a whole.

The second thing that we'll introduce in this chapter is some of the library and framework code that can make our lives easier. Using a raw XHR object, we wrote a lot of code to figure out various cross-browser conundrums, and manually orchestrated all the fine-grained details of the HTTP request and response. This was fine for learning how HTTP works, but when we're writing production code, we will generally be concerned with higher-level issues regarding application state and logic, and have the low-level plumbing details taken care of for us.

Fortunately, several good frameworks and libraries are available for Ajax, and we'll lean on some of them here. We'll present some of the frameworks more formally in chapter 4. For now, we'll introduce them as we go along.

In the examples in this chapter, we want to focus on the way in which data is passed between the client and the server. We'll therefore be sticking with the "Hello World" type of example throughout this chapter. The examples in this chapter use JavaServer Pages (JSP) on the back end, and have been tested on a Tomcat web server. We provide a `.war` file for all the examples in the download code that accompanies this book, which provides a launching page for the various examples, as illustrated in figure 1.5.

Now, without further ado, let's take a look at the first example of using Ajax to communicate with the server.

Although it is quite straightforward to create an Ajax request as we have seen in the previous section, there's a lot of bookkeeping involved. When we're delivering a real web application to a client, asynchronous calls to the server are simply a means to an end. Having to focus on `readyStates`, HTTP headers, and URL-encoded query strings every time we want to talk to our domain model will be tedious and distracting.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

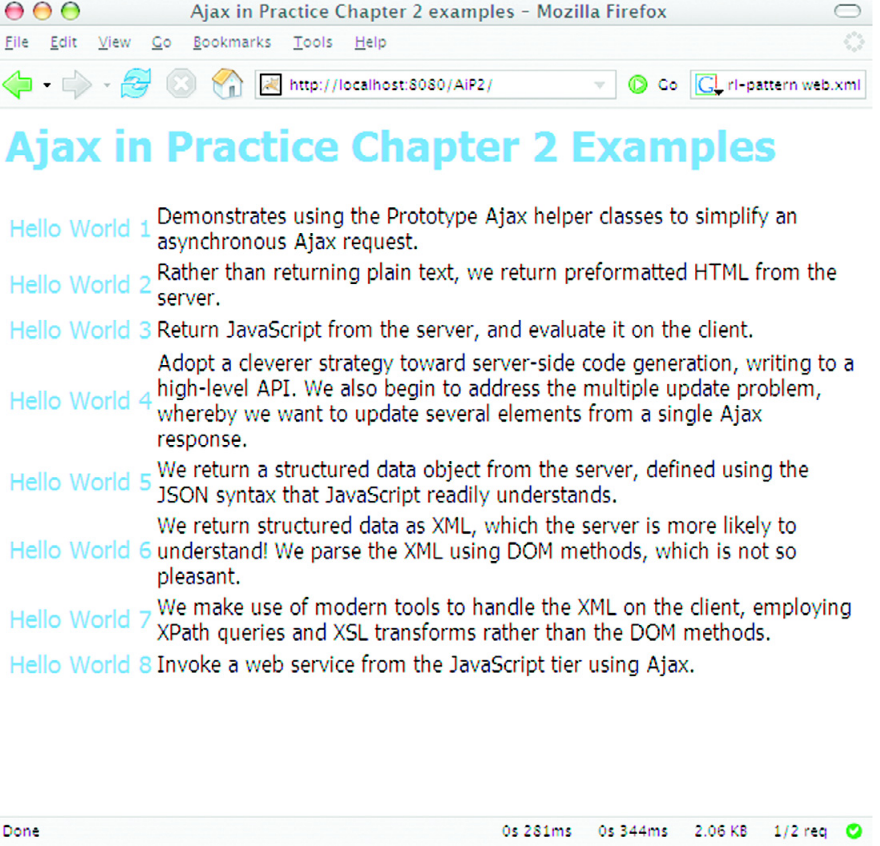

Figure 1.5 Launch page for the examples in chapters 1 and 2

As with any software development, having understood the issues involved in a particular task, we want to encapsulate our solutions into a helper object or set of functions that allows us to focus our attention on the next level up. We want to be able to set up a standard Ajax request and handle the response in as few lines of code as possible, and still have access to the fine-tuning capabilities when we need them.

We could write our own helper library to encapsulate the XHR object, or we could make use of a third-party library that has already done the grunt work for us. In this section, we'll look at the Prototype library's Ajax helper classes, and see how they can simplify Ajax for us. First, we'll look at the business of making a request.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

1.3.1 *Making requests with Prototype's Ajax.Request object*

When we sent an Ajax request to the server in our example in section 1.2, we faced a number of messy issues. First, we had to create an XHR object in a browser-independent fashion. Second, we had to invoke several methods on the XHR object to provide it with a URL, HTTP method, and POST body, and set other HTTP headers. Getting these right required a working knowledge of the HTTP protocol. A working knowledge of the underlying stack is always a good thing, but we shouldn't be forced to think about it every time we make a call back to the server.

A good wrapper object such as Prototype's Ajax.Request will automate the cross-browser issues for us. It will also allow us to pass in only the information that concerns us, and automatically provide sensible defaults for any parameters that we don't explicitly provide.

Problem

Working with the XHR object requires us to concentrate on a lot of low-level details, such as obtaining the object in a cross-browser fashion, and responding to subtle changes in `readyState` during the arrival of the response.

Solution

Use a framework to simplify creating an Ajax request. Let's start by introducing our simple application, the UI for which is shown in figure 1.6.

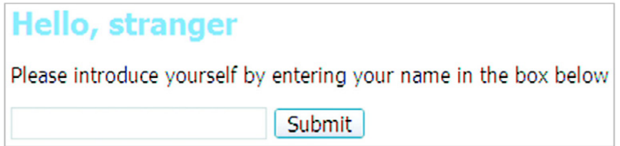

Figure 1.6
User interface for version 1 of our Hello World application

We've provided a text input box and a submit button. When the button is clicked, the text in the input box will be sent to the server, and the phrase "Hello, stranger" replaced with the name returned by the server. In this case, the server isn't actually doing anything with the name—it's simply echoing it back—but our concern here is fielding the response on the client. On the server, any kind of processing might be taking place. We'll start by looking at the client-side code.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Coding the client

Listing 1.2 shows the client-side code for this example.

Listing 1.2 hello1.html

```
<html>
<head>
<title>Hello Ajax version 1</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; font-size: 1.5em; }
</style>
<script type='text/javascript'
  src='prototype.js'> </script>                      ❶ Includes Prototype library
<script type='text/javascript'>
window.onload=function() {
  document.getElementById('helloBtn')
  .onclick = function(){
    var name = document.getElementById('helloTxt')
      .value;
    new Ajax.Request(                                 ❷ Creates Ajax.Request object
      "hello1.jsp?name = "+encodeURI(name),           ❸ Provides URL (mandatory)
      {
        method:"get",                                 ❹ Provides optional parameters
        onComplete:function(xhr){
          document.getElementById('helloTitle')
          .innerHTML = xhr.responseText;
        }
      }
    );
  };
};
</script>
</head>
<body>
<h1 id='helloTitle'>Hello, stranger</h1>
<p>Please introduce yourself by entering
your name in the box below</p>
<input type='text' size='24' id='helloTxt'>
</input> 
<button id='helloBtn'>Submit</button>
</body>
</html>
```

The first thing that we need to do is to include the Prototype library in our page ❶. Prototype ships as a single file, which makes this a one-line task.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In the `window.onload` event handler, we reference the button object, and assign a click event handler to it. Within the event handler, we read the name from the text input box ❷, and then create an Ajax.Request object. This object takes two arguments. The first is the URL of the resource on the server ❸. The second is a free-form JavaScript object that may contain an arbitrary set of extra configuration options ❹. These can include HTTP verbs, headers, request bodies, a variety of callback options and other features. In this case, we only pass in two options. First, we set the HTTP method to GET, because the Ajax.Request defaults to POST. Second, we provide a callback function.

Looking back at section 1.3, we saw that the XHR object took a callback that was triggered whenever the `readyState` was changed. When the Ajax.Request creates an XHR object internally, it defines an internal callback to handle these fine details. As an end user of the library, we can provide higher-level callbacks, such as `onComplete`, that fit better with our immediate requirements. In the relatively rare cases that we do want to be notified of other changes in `readyState`, we can provide additional callback functions to capture these.

Our callback function is admirably simple, modifying the text in the title element of the page, as shown in figure 1.7. The user has typed in their name and clicked the submit button, and the title has been modified.

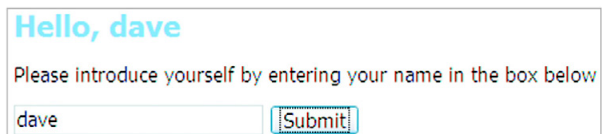

Figure 1.7 Hello World version 1 after processing the Ajax response

That's the client-side code handled for this example. In the next section, we'll take a brief look at the server-side code.

Coding the server

The server-side implementation for this example is extremely simple, as you can see in listing 1.3.

Listing 1.3 hello1.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
%>
Hello, <%=name%>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

As we noted earlier, we've made the server-side code as simple as possible for this example, because we want to focus on the client-side code. Please feel free to imagine a complex n-tier app sitting behind this simple JSP—the principles are the same.

Discussion

The Ajax.Request class provided by Prototype allowed us to make our Ajax request with a minimum of fuss. All we had to provide was a URL, the HTTP method, and a callback function. Internally, Ajax.Request worked out how to create the XHR object and filled in the blanks when shaping the request. It also simplified the callback semantics considerably, allowing us to supply a function that would only be called once on completion of the response, and that therefore only had to deal with application logic.

Prototype provides a lot more than just the Ajax helper classes. In this example, we've deliberately avoided using any of these other features, as we're simply using Ajax.Request as an example of a well-designed wrapper object. Similar wrappers, with equally straightforward calling semantics, exist in other popular Ajax libraries. Dojo has the dojo.io.Request class, MochiKit has MochiKit.Async, and jQuery has the `$.ajax()` function, to name but three. Depending on the library that you plan to use, the exact capabilities of your Ajax wrapper will vary, but you're likely to experience a satisfactory time savings from any of them.

The callback function that we provided to handle the response was, in this case, very simple. In the next example, we'll see what's needed to allow for Ajax-based generation of richer content.

1.3.2 Simplifying Ajax responses

In the previous example, we used the `innerHTML` property to write the server contents directly into a DOM element on the page. The response that the server offered us was a simple piece of text. In many cases, though, we will want the server to deliver more complicated information to us, often with an internal structure of its own. In chapter 2, we'll look at ways of passing that structured information between the server and the client.

Structured data lies at the heart of most web applications. On the server side, the data is typically stored in a relational database. On the client, the data is displayed as some sort of report or user interface. In between, the data may be manipulated or operated on by the logic of the application. Traditionally, this has been done entirely on the server, with the client acting as a dumb terminal. With

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Ajax, we have the capability to run logic on the client or the server, or even both. There are many permutations to explore.

We'll start off by following the dumb terminal approach. The server generates a rich report from the application data, and sends it to the client for display. The client doesn't need to know what the information means, but only how to display it. We'll jazz up the response a little for this example, returning an extra bit of content (entirely trivial, in this case!) below the header. The example with the full response inserted into the page is shown in figure 1.8.

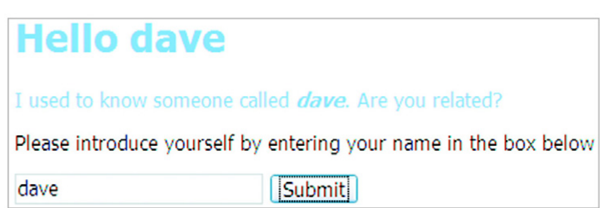

Figure 1.8 Result of returning rich HTML in the Ajax response

Problem

The server is sending us a rich visual report (rather than a simple piece of text) on the application state that is to be incorporated into our user interface.

Solution

The solution to this problem is quite straightforward, and requires little in the way of JavaScript coding. The changes on the client are minimal, in fact, and we can still use the `innerHTML` property to paste the response into our document. Listing 1.4 shows the client-side code for example 2.

Listing 1.4 hello2.html

```
<html>
<head>
<title>Hello Ajax version 2</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
    var name = $('helloTxt').value;
    new Ajax.Request(
      "hello2.jsp?name = "+encodeURI(name),
      {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          method:"get",
          onComplete:function(xhr){
            $('helloTitle').innerHTML = xhr.responseText;
          }
        }
      );
    };
  };
</script>
</head>
<body>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button
</body>
</html>
```

The main changes that need to be made here are on the server. Our modified JSP is shown in listing 1.5.

Listing 1.5 hello2.jsp

```
<jsp:directive.page contentType="text/html"/>
<%
String name=request.getParameter("name");
%>
<h1>Hello <%=name%></h1>
<p>I used to know someone called <b><i><%=name%></i></b>. Are you
related?</p>
```

We've set the MIME type of the response to `text/html`, out of politeness. This isn't strictly necessary but provides a statement of intent. Some Ajax libraries will pay attention to the MIME type of responses, as you'll see, so it's a good habit to get into.

The body of the response generates a small bit of content from the input parameters. In this case, we're not doing anything other than parroting back the name supplied by the user, but that's just to keep the back end simple for our example. In production, we'd have some proper code—front controllers, domain objects, databases, and so on—taking the request parameters and generating a

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

response. As far as the Ajax mechanics are concerned, the effect would be the same, with a fragment of HTML being returned to the client.

Discussion

Generating HTML fragments on the server is a viable strategy for Ajax web app development, and can provide everything one needs. It offers a clear incremental migration path from classic web applications and frameworks, in which the server is already generating HTML to send to the browser.

We used `innerHTML` to insert the new fragments into the existing page. This is effective in the majority of cases, completely wiping out the existing content in an element and replacing it with the new content. If we want finer control, there are other DOM methods such as `insertAdjacentHTML()` and `createContextual-Fragment()`, but these are generally not cross-browser. The Prototype library offers cross-browser wrappers around these in the form of the Insertion objects, and other libraries may provide similar features.

When using `innerHTML` , there are a couple of "gotchas" you need to be aware of. First, `<script>` tags included in a page using `innerHTML` will be ignored by the browser. Second, HTML table elements, with the exception of the individual cell (i.e., the `<TD>` tag) have read-only `innerHTML` properties.

Finally, it's worth pointing out that the Prototype library provides a special subclass of Ajax.Request that makes it even easier to do this sort of thing by automatically assigning the response as the `innerHTML` property of a named DOM element. If you're using Prototype and want to send HTML fragments from your server, we urge you to take a look at the Ajax.Updater, but we didn't want to use it here ourselves because it hides a lot of the underlying mechanics that we wished to explore, and it would focus the discussion too narrowly on a single library.

1.4 Summary

We began this chapter with a brief discussion of where Ajax is now. The technology is maturing fast, and doing some interesting things. In terms of general trends, we noted two things. First, the discussion has moved on from how to make an Ajax request to what to do with this capability. Interesting issues are being raised in terms of web application architecture, and in terms of business models, and what new possibilities Ajax is enabling.

The second point to note is the emergence of mature frameworks and libraries for Ajax. The days of hand-coding cross-browser plumbing are, thankfully,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

receding, and many good practices are being wrapped up in libraries such as Prototype, Dojo, Rico, and DWR, to name a few.

These two issues shape this book. We want to discuss the higher-level issues around Ajax development, and we want to show you how to make life easy when addressing these issues by using best-of-breed toolkits. Together, these will give you the practical knowledge to use Ajax successfully in real-world settings.

We can't forget the plumbing and low-level details entirely, though. In the second part of this chapter, we walked through the mechanics of the XHR object. We then showed you how to make it a lot easier by using a wrapper object, in this case Prototype's Ajax.Request. In the final example, we began to address the next level of application design, by asking what sort of data the server might respond with. We looked at the simplest approach, whereby we retrieved fragments of HTML from the server, and stitched them into the existing page.

This approach can serve us well, and deliver a lot of the benefits of Ajax to our apps with relatively simple JavaScript. However, it cannot provide a high degree of responsiveness to complex operations on the client, as all-important decisions require a round-trip to the server. For that, we need to move some of the intelligence to the client. In the next chapter, we'll look at ways of doing that, using JavaScript, JSON, and XML.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

2 How to talk Ajax

This chapter covers

- Identifying the main dialects of Ajax
- Using Ajax with JavaScript and JSON
- Working with XML, XPath, and web services

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Ajax is a melting pot, with many different approaches to application design and a myriad of dialects. In chapter 1 we looked at the mechanics of the XHR object that lies at the heart of Ajax, and you saw how to wrap those details up in a helper object. Without the distraction of having to write all that plumbing code by hand, we can focus on the more interesting issues of structuring the communication between the server and the client. We can now look at the different categories of Ajax communication, and teach you how to talk Ajax in a range of dialects.

We'll continue to work with the Hello World example that we introduced in chapter 1 and the problem/solution format. We'll also continue to use frameworks to handle the low-level Ajax concerns for us and free us up to look at the interesting issues. Many of the examples will continue to use Prototype's Ajax. Request object, but we'll also take a look at Sarissa and a web services client toolkit. Let's begin by looking at JavaScript as a medium of communication.

2.1 Generating server-side JavaScript

When the server returns HTML from an Ajax request, we can generate complex user interfaces on the fly, but they remain largely static. Any significant interaction with the application will require further communication with the server. In many cases, this isn't a problem, but in others, it is necessary to deliver behavior as well as content. All client-side behavior is driven by JavaScript, so one way forward is for the server to generate JavaScript for us.

2.1.1 Evaluating server-generated code

When handling server-generated HTML, we can go a long way using only `innerHTML`. When handling server-generated JavaScript, we can make similar use of the `eval()` method. JavaScript is an interpreted language, and any snippet of text is a candidate for evaluating as code. In the next example, we'll see how to use `eval()` as part of the Ajax-processing pipeline.

We'll stick with our Hello World app through this chapter. In this first example, we'll use the response to modify the title element again, as shown in figure 2.1.

Hello, *dave*

Please introduce yourself by entering your name in the box below

dave Submit

Figure 2.1 Result of evaluating server-generated JavaScript

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

The server is returning JavaScript code from an Ajax request. We need to run the code when we receive it.

Solution

Using `eval()` is almost as simple as using `innerHTML`. Listing 2.1 presents the third incarnation of our Hello World application.

Listing 2.1 hello3.html

```
<html>
<head>
<title>Hello Ajax version 3</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick=function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello3.jsp?name="+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          eval(xhr.responseText);        <-- 1 Evaluates the response
        }
      }
    );
  };
};
</script>
</head>
<body>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
 in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</body>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Comparing our earlier solution with listing 2.1, you can see that we've had to change very little. We can still read the response body using the `responseText` property of the XHR object, which we then pass straight to `eval()` ❶.

We want to modify the title block of the page when the response comes in. To do this, we need to perform a bit of DOM manipulation. The method calls can be generated directly on the server, as shown in listing 2.2.

Listing 2.2 hello3.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
%>
document.getElementById('helloTitle').innerHTML =
  "<h1>Hello, <b><i>"+name+"</i></b></h1>";
```

Generally, it is good manners to set the MIME type, but we've switched it off here because Prototype is clever enough to recognize the `text/javascript` MIME type and would evaluate it automatically for us. Here we want to do the evaluation manually in order to demonstrate some general principles, not show off Prototype's power-user features!

Discussion

In this example, we've demonstrated the principle of passing JavaScript from the server to the client, but in the process we've raised a few interesting problems. We'll fix these up in the next example, but first let's examine them.

The first problem is that we've created a very tight coupling between the client and server code. The JSP needs to know the `id` attribute of the DOM element that it is going to populate. If we change the HTML on the user interface, we need to alter the server code. In a small example like this one, that's not too great a burden, but it will quickly become unscalable.

Second, we've created a solution looking for a problem. We aren't doing anything here that we couldn't do more elegantly and simply using `innerHTML`. As long as we're using the response to update a single element on the page, this approach is overkill.

In the next example, we're going to address both these points and see how to reduce the coupling across the tiers as well as update several elements at once.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

2.1.2 Utilizing good code-generation practices

When we generate JavaScript on the server, we are practicing code generation. Code generation is an interesting topic in its own right, with a well-established set of conventions and ground rules. A cardinal rule of code generation is to always generate code at the highest level possible.

In the next example, we're going to increase the complexity of our Hello World application a little and demonstrate how to tighten up our code generation to cope with it.

Problem

Writing low-level JavaScript on the server leads to unacceptable tight coupling between the server and client codebases. This will give our application severe growing pains and lead to increased brittleness.

At the same time, we want to maintain a list of previous visitors to our page, as well as display the name of the current visitor. The server is going to classify visitors' names as either long or short, and we'll provide a separate list for each (once again, this is a surrogate for real business logic, because we want to keep the server code simple in this chapter).

We'll also pop up an alert message when the data comes in. The revised UI for the Hello World app is shown in figure 2.2.

Every time the form is submitted to the server, the most recent name will be displayed in the title element, as before. We'll also keep a running list of visitors in the box elements on the left. Figure 2.3 shows our version 4 Hello World in action, after several interesting visitors have passed by!

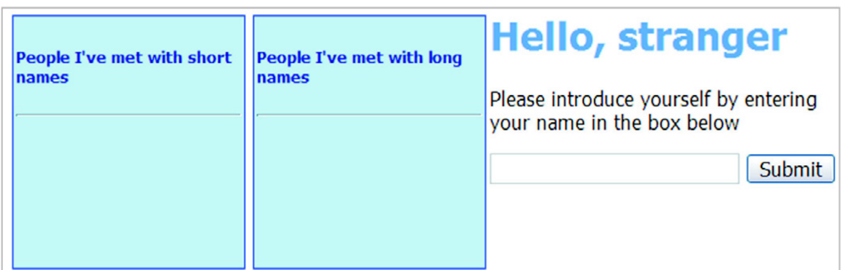

Figure 2.2 Expanded UI for version 4 of Hello World, with a list of previous visitors alongside the form

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

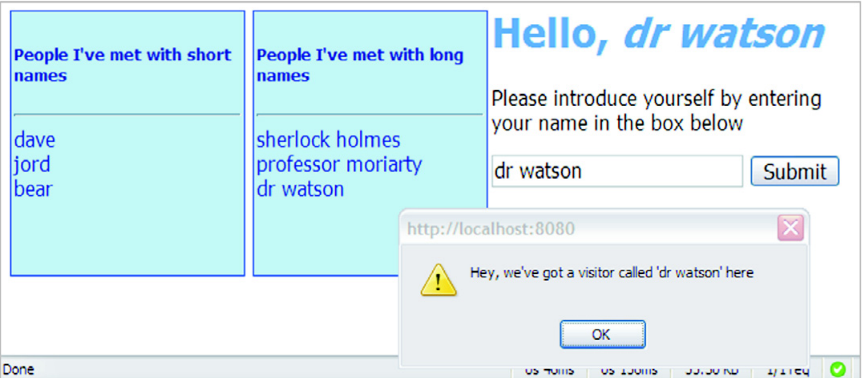

Figure 2.3 Hello World version 4 after several visits. Here we see a modified title, an updated list to the left, and an alert message, all from a single server-generated call.

Solution

When the response comes back from the server, we want to update the client with the new information. The code that the server is sending us is simply a carrier for some data, so we will simplify the server-generated JavaScript to call a single `updateName()` function, passing in the data as arguments.

On the client side, we need to define that `updateName()` function as handwritten JavaScript, as shown in listing 2.3. `updateName()` will handle all of our expanded requirements.

Listing 2.3 hello4.html

```
<html>
<head>
<title>Hello Ajax version 4</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello4.jsp?name = "+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          eval(xhr.responseText);              <-- Evaluates response
        }
      }
    );
  };
};

function updateName(name,isLong){              <-- Defines API
  $('helloTitle').innerHTML=
    "<h1>Hello, <b><i>"+name+"</i></b></h1>";
  var listDivId=(isLong)
    ? 'longNames' : 'shortNames';
  $(listDivId).innerHTML+=name+"<br/>";
  alert("Hey, we've got a visitor called '"
    +name+"' here");
}

</script>
</head>
<body>

<div id='shortNames' class='sidebar'>
<h5>People I've met with short names</h5><hr/>
</div>
<div id='longNames' class='sidebar'>
<h5>People I've met with long names</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
 in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In spite of the increased complexity of our requirements, the server-side code has become simpler. Listing 2.4 shows the JSP used to serve data to version 4 of our app.

Listing 2.4 hello4.jsp

```
<jsp:directive.page contentType="text/plain"/>
<%
String name = request.getParameter("name");
boolean isLong = (name.length() > 8);
%>
updateName("<%= name %>",<%= isLong %>);
```

The JSP is simpler in terms of length of code, but also in the number of concepts. It is only concerned with the business logic appropriate to the server and talks to the client via a high-level API.

Discussion

With this example, we've crossed an important threshold and need to update multiple regions of the UI from a single Ajax call. At this point, the simple `innerHTML` approach that we used in example 2 can no longer suffice. In this case, the requirements were somewhat artificial, but in many real-world applications, multiple update requirements exist. For example:

- In a shopping cart, adding a new item will result in adding a row to the cart body, and updating the total price, and possibly shipping costs, estimated shipping date, and so on.
- Updating a row in a data grid may require updates to totals, paging information, and so on.
- A two-pane layout in which a summary list is shown on the left and drill-down details of the selected item on the right will have a tight interdependency between the two panes.

We solved the multiple-update issue in this case by defining a JavaScript API and generating calls against that API. In this case, we defined one API method and called it only once, but a more sophisticated application might offer a handful of API calls and generate scripts consisting of several lines. As long as we stick to the principle of talking in conceptual terms, not in the details of DOM element IDs and methods, that strategy should be able to work for us as the application grows.

An alternative to generating API calls on the server is to generate raw data and pass it to the client for parsing. This opens up a rich field, which we'll spend the remainder of this chapter exploring.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

2.2 Introducing JSON

We began our exploration of Ajax techniques in chapter 1 by doing all the processing on the server and sending prerendered HTML content to the browser. In section 2.1, we looked at JavaScript as an alternative payload in the HTTP response. The crucial win here was that we were able to update several parts of the screen at once. At the same time, we were able to maintain a low degree of coupling between the client-side and server-side code.

If we follow this progression further, we can divide the responsibilities between the tiers, such that only business logic is processed server-side and only application workflow logic on the client. This design resembles a thick-client architecture, but without the downside of installing and maintaining the client on client PCs.

In this type of design, the server would send data to the client—potentially complex structured data. As we noted at the beginning of this chapter, we have a great deal of freedom as to what form this data can take. There are two strong contenders at the moment: JavaScript Object Notation (JSON) and XML. We'll begin to explore data-centric Ajax in this section with a look at JSON.

A one-minute JSON primer

Before we dive into any examples, let's quickly introduce JSON. JSON is a lightweight data-interchange format that can be easily generated and parsed in many different server-side technologies and in JavaScript. A complete data-interchange format will provide two-way translation between the interchange format and live objects, as illustrated in figure 2.4.

Half of JSON is provided for free as part of the JavaScript language specification, and the other half is available as a third-party library. That sounds like an unusual state of affairs, so let's explain what we mean by it.

First, let's look at what a JSON definition looks like. The following example defines a variable `customers` and stores in it an array attribute called `details`. Each array element is a collection of attributes of each customer object. Each customer object has three attributes: `num`, `name`, and `city`.

```
var customers = { "details": [
               {"num": "1","name":"JBoss","city":"Atlanta"},
               {"num": "2","name":"Red Hat","city":"Raleigh"},
               {"num": "3","name":"Sun","city":"Santa Clara"},
               {"num": "4","name":"Microsoft","city":"Redmond"}
            ]
         } ;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

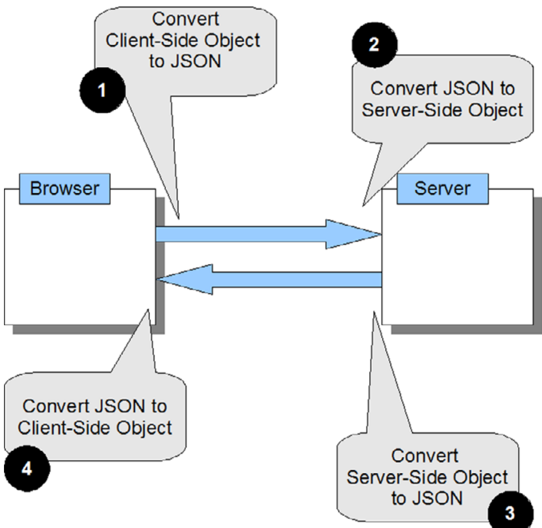

Figure 2.4 JSON as a round-trip data-interchange format

We've defined this rather complex variable using JSON syntax. At the same time, all we've written is a standard piece of JavaScript. Curly braces are used to delimit JavaScript objects (which behave kind of like associative arrays), and square braces delimit JavaScript Array objects. If you want to brush up on your core JavaScript language skills, we cover these things in greater depth in chapter 4.

Once we've defined the variable, we can easily read its values using standard JavaScript syntax:

```
alert (customers.details[2].name);
```

This would display the string "Sun" in an alert box. So far, all we've done is take a standard piece of JavaScript syntax and called it JSON.

We can also create a string variable and then evaluate it using `eval()` to generate our variable:

```
var customerTxt = "{ 'details': [ "  +
              "{'num': '1','name':'JBoss','city':'Atlanta'}, " +
              " {'num': '2','name':'Red Hat','city':'Raleigh'}, " +
              " {'num': '3','name':'Sun','city':'Santa Clara'}, " +
              " {'num': '4','name':'Microsoft','city':'Redmond'}" +
            "] }" ;
var cust = eval ('(' + customerTxt + ')');
alert (cust.details[0].city); //shows 'Atlanta'
```

There's no good reason to write code like this when we're declaring the string ourselves, but if we're retrieving the string in a different way—say, as the response

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

to an Ajax request—then suddenly we have a nifty data format that can express complex data structures easily and that can be unpacked with extreme ease by the JavaScript interpreter.

At this point, we have half a data-interchange format. Standard JavaScript doesn't provide any way of converting a JavaScript object into a JSON string. However, third-party libraries can be found at http://www.json.org, which allow us to serialize client-side objects as JSON, using a function called `stringify()`. The JSON library also provides a `parse()` method that wraps up the use of `eval()` nicely.

You'll also find libraries at json.org for creating and consuming JSON in a number of server-side languages. With these tools, it's possible for the client and server to send structured data back and forth as JSON over the entire course of a web application's user session.

Let's return to our Hello World example for now, and see how the client handles a JSON response.

2.2.1 Generating JSON on the server

We can go quite a long way with JSON, so let's break it up into two parts. First, we're going to look at how far we can get simply by using the browser's built-in ability to parse JSON data, and replace the generic JavaScript response from the previous example with a JSON object definition.

Problem

We want the server to respond to our request with rich structured data, and let the client decide how to render the data.

Solution

Sticking with the Hello World theme, this example is going to return a fuller description of the individual than just their name:

- The person's initial, calculated on the server using string manipulation
- A list of things that the person likes
- Their favorite recipe, encoded as an associative array

Figure 2.5 shows the application after receiving a response.

In keeping with previous examples, the back end is going to be pretty dumb and will, in fact, return the same data (apart from the initial) for every name. It's a simple step from dummy data to a real database, but we don't want to confuse things by introducing too many Java-specific back-end features, as the client-side code can talk to any server-side technology.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

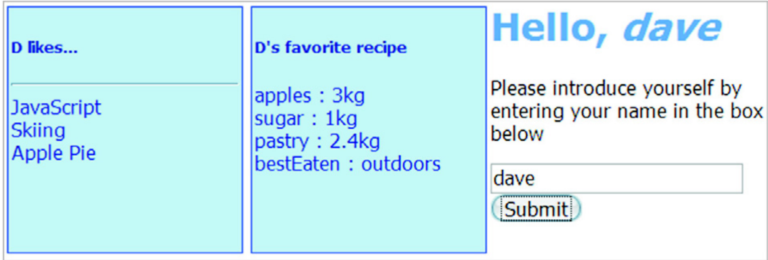

Figure 2.5 JSON-powered Hello World application displaying rich data

So, first we're going to do things the simple way and just make use of JavaScript's built-in JSON-parsing capabilities. Our client-side code appears in listing 2.5.

Listing 2.5 hello5.html

```
<html>
<head>
<title>Hello Ajax version 5</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello5.jsp?name = "+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          var responseObj = eval("("+xhr.responseText+")");   <-- (1) Parses JSON response
          update(responseObj);
        }
      }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      );
    };
  };

  function update(obj){
    $('helloTitle').innerHTML = "<h1>Hello, <b><i>"
      +obj.name
      +"</i></b></h1>";
    var likesHTML = "<h5>"
      +obj.initial
      +"likes...</h5><hr/>";
    for (var i=0;i<obj.likes.length;i++){
      likesHTML += obj.likes[i]+"<br/>";
    }
    $('likesList').innerHTML = likesHTML;
    var recipeHTML = "<h5>"
      +obj.initial
      +"'s favorite recipe</h5>";
    for (key in obj.ingredients){
      recipeHTML += key
       +" : "
       +obj.ingredients[key]
       +"<br/>";
    }
    $('ingrList').innerHTML=recipeHTML;
  }

  </script>
  </head>
  <body>

  <div id='likesList' class='sidebar'>
  <h5>Likes</h5><hr/>
  </div>
  <div id='ingrList' class='sidebar'>
  <h5>Ingredients</h5><hr/>
  </div>
  <div>
  <div id='helloTitle'>
  <h1>Hello, stranger</h1>
  </div>
  <p>Please introduce yourself by entering your name
     in the box below</p>
  <input type='text' size='24' id='helloTxt'></input>
   
  <button id='helloBtn'>Submit</button>
  </div>
  </body>
  </html>
```

❷ Uses parsed object

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Working with JSON in this way is pretty simple. We use `eval()` to parse the JSON response ❶, remembering to add parentheses around the string before we parse it. Using the parsed object in our `update()` method ❷ is then entirely natural, because it's just another JavaScript object.

Let's look briefly at the server-side code required to get us this far. Listing 2.6 shows iteration 5 of our JSP file.

Listing 2.6 hello5.jsp

```
<jsp:directive.page contentType="application/javascript"/>
<%
String name=request.getParameter("name");
%>
{
  name: "<%=name%>",
  initial: "<%=name.substring(0,1).toUpperCase()%>",
  likes: [ "JavaScript", "Skiing", "Apple Pie" ],
  ingredients: {
    apples: "3kg",
    sugar: "1kg",
    pastry: "2.4kg",
    bestEaten: "outdoors"
  }
}
```

As we said earlier, most of the data that we've generated here is dummy data. What's interesting to us here is the creation of the JSON string, which we've simply written out by hand, inserting variable values where appropriate.

Discussion

We've demonstrated in this example that parsing JSON on the client is extremely easy, and that alone makes it a compelling possibility. However, looking back at figure 2.4, you can see that we've only covered one of the four stops in the full round-trip between client and server: the conversion of JSON to client-side objects. For a small app like this one, what we've done so far is good enough, but in larger apps, or those handling more complex data, we would want to automatically handle all aspects of serialization and deserialization, and be free to concentrate on business logic on the server and rendering code on the client. Before we leave JSON, let's run through one more example, in which we execute a full round-trip between the client and server.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

2.2.2 Round-tripping data using JSON

When we're writing the client callback, we love JSON, because it makes everything so simple. However, we passed the request data down to the server using a standard HTTP query string, and then constructed the JSON response by hand. If we could manage all communication between the browser and server using JSON, we might save ourselves a lot of extra coding.

To get to that happy place, we're going to have to employ a few third-party libraries. So, let's get coding, and see how happy we are when we've got there.

Problem

We want to apply JSON at all the interfaces between our application tiers and HTTP, so that the client code can be written purely as JavaScript objects and the server purely as Java (or PHP, .NET, or whatever) objects.

Solution

We can use figure 2.4 as a crib sheet, to see where the gaps in our design are. On the browser, we've already handled step 4, the conversion of the response text into a JavaScript object. We still need to consider the conversion of the object into JSON on the client, though. To do this, we'll need to use the json.js library from www.json.org. Listing 2.7 shows how it works.

Listing 2.7 hello5a.html

```
<html>
<head>
<title>Hello Ajax version 5a</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript' src='json.js'> </script>    <-- (1) Includes JSON library
<script type='text/javascript'>
window.onload = function(){
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  $('helloBtn').onclick = function(){
    var name = $('helloTxt').value;
    new Ajax.Request(
      "hello5a.jsp",
      {
        postBody:JSON.stringify({name:name}),          ❷ Converts object to JSON
        onComplete:function(xhr){
          var responseObj = JSON.parse(xhr.responseText);     ❸ Converts JSON to object
          update(responseObj);
        }
      }
    );
  };
};

function update(obj){
  $('helloTitle').innerHTML = "<h1>Hello, <b><i>"+obj.name+"</i></b></h1>";
  var likesHTML = "<h5>"+obj.initial+" likes...</h5><hr/>";
  for (var i=0;i<obj.likes.length;i++){
    likesHTML+=obj.likes[i]+"<br/>";
  }
  $('likesList').innerHTML=likesHTML;
  var recipeHTML="<h5>"+obj.initial+"'s favorite recipe</h5>";
  for (key in obj.ingredients){
    recipeHTML+=key+" : "+obj.ingredients[key]+"<br/>";
  }
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The first thing that we need to do is include the json.js library ❶. Once we've done that, we can simplify the response-handling code by using the `JSON.parse()` method ❸. More importantly, though, we can reconsider the way we put together the request.

So far, we've been sending GET requests to the server, passing in data on the query string. This is fine for requesting data, but when we want to update information or send a more complex request to the server, we'd be better off using a POST request. POST requests have a body as well as a set of headers, and we can populate that body with any text that we want. Here, we're going to use JSON.

We're still using Prototype to send the request, and we now pass in a `postBody` property with the options to the Ajax.Request constructor. The value of this is the result of calling `JSON.stringify()` ❷. `stringify()` takes a JavaScript object as an argument and recurses through it, writing it out as JSON. Thus, our POST body will not contain URL-encoded key-value pairs, as it would if sent from an HTML form, but a JSON string, something like this:

```
{ name: 'dave' }
```

For such a simple piece of structured data, this might be considered overkill, but we could potentially pass very complex data in this way.

Now that we've figured out the client side of the solution, let's turn to the server. We happen to be using Java on the server for these examples, and Java knows nothing about JSON whatsoever. Neither do most server-side languages. So, to make sense of the response we've just been sending, we'll need to bring in a third-party library.

Whatever your server-side technology, you're likely to find a JSON library to fit it at www.json.org (scroll down to the bottom of the page). We selected Json-lib, which is based on Doug Crockford's original JSON for Java libraries.

Json-lib has quite a bit of work to do. JSON encodes structured data in a very fluid way, and Java is a strongly typed language, so the two don't sit together naturally. Nonetheless, we managed to get the job done without too much trouble. Listing 2.8 shows the not-so-gory details.

Listing 2.8 hello5a.jsp

```
<jsp:directive.page
  contentType="application/javascript"
  import="java.util.*,net.sf.json.*"      ❶ Imports JSON classes
/>
<%
String json=request.getReader().readLine();     ❷ Reads POST body
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
JSONObject jsonObj=new JSONObject(json);            ❸ Parses JSON string
String name=(String)(jsonObj.get("name"));          ❹ Reads parsed object

jsonObj.put("initial",
  name.substring(0,1).toUpperCase());               ❺ Adds new values

String[] likes=new String[]
  { "JavaScript", "Skiing", "Apple Pie" };
jsonObj.put("likes",likes);

Map ingredients=new HashMap();
ingredients.put("apples","3kg");
ingredients.put("sugar","1kg");
ingredients.put("pastry","2.4kg");
ingredients.put("bestEaten","outdoors");
jsonObj.put("ingredients",ingredients);             ❻ Writes object as JSON
%><%=jsonObj%>
```

In order to use the Json-lib classes in our project, we need to import the net.sf.json package, which we do in the `<jsp:directive.page>` tag ❶. Now, on to the code.

The first challenge that we face is decoding the POST body. The Java Servlet API, like many web technologies, has been designed to make it easy to work with POST requests sent from HTML forms. With a JSON request body, we can't use `HttpServletRequest.getParameter()`, but need to read the JSON string in the request via a `java.io.Reader` ❷. Similar capabilities are available for other technologies. If you're using PHP, use the `$HTTP_RAW_POST_DATA` variable. If you're using the .NET libraries, you'll need to get an `InputStream` from the HttpRequest object, much as we've done here with our Java classes.

Back to the Java now. Once we've got the JSON string, we parse it as an object ❸. Because of the fundamental disjoint between loosely typed JSON and strictly typed Java, the Json-lib library has defined a JSONObject class to represent a parsed JSON object. We can read from it using the `get()` method ❹ and extract the name from the request.

Now that we've deserialized the incoming JSON object, we want to manipulate it, and then send it back to the client again. The JSONObject class is able to consume simple variable types such as strings, arrays, and Java Maps (that is, associative arrays) ❺, to add extra data to the object. Once we've modified the object, we serialize it again ❻, sending it back to the browser in the response.

And that's it! We've now sent an object from the client, modified it on the server, and returned it back to the client again.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

This has been the most complex example so far, demonstrating a way of communicating structured objects back and forth over the network on top of the text-based HTTP protocol. Because both the client and server can understand the JSON syntax, with a little help from some libraries, we haven't had to write any parsing code ourselves. However, as we noted, JSON is suited for use with loosely typed scripting languages, and so there was still some translation required. The goal of a system like this is to be able to serialize and deserialize our domain objects over the network. If our domain objects graph is written in Java (or C#, say), then we still need to manually translate them into generic hashes and arrays before they can be passed to the JSON serializer. The clumsiest piece of coding in our round-trip was in the JSP, where we assembled the Maps and lists of data for the JSONObject. This problem is strongly emphasized in the case of Java, which lacks a concise syntax for defining associative arrays in particular, compared with Ruby or PHP, for example.

From square brackets to angle brackets

There is another text-based format that can be understood by both client and server: XML. Most server-side languages have good support for XML, so we might find that we have an easier time working with XML than with JSON on the server. In the next section, we'll look at XML and Ajax, and see whether that is the case.

2.3 Using XML and XSLT with Ajax

XML is a mature technology for representing structured data, supported by most programming languages either as a core part of the language or through well-tested libraries or extensions. In the remainder of this chapter, we'll look at how various XML technologies work with XML, and complete our survey of basic Ajax communication techniques.

The XMLHttpRequest object that we've been using for our Ajax requests has special support for XML built into it. So far, we've been extracting the body of the HTTP response as text and parsing it from there. JavaScript in the web browser doesn't have a standard XML parser available to it, but the XHR object can parse XML responses for us, as we'll see in the next example.

2.3.1 Parsing server-generated XML

So far, we've had the server generate HTML, JavaScript, and JSON responses for us in various versions of our Hello World application. All of these formats are

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

designed to appeal to the browser rather than the server. XML, in contrast, is often used to communicate between server processes, in a variety of technologies ranging from RSS syndication feeds to web service protocols, such as XML-RPC and SOAP. If we're transmitting information from our domain objects up to the client, then many server-side technologies provide support for serializing and deserializing objects as XML.

In any of these scenarios, we may find that it's easy to transmit data as XML, from the perspective of the server. If this is going to be a useful way forward, then we'll also need to handle the XML on the client. We'll start by looking at the built-in support for XML offered by the XHR object. XML, like JSON, is a format for exchanging structured data. The best way to compare the two is to set them the same task, so we'll follow the lead from the previous section and supply a list of likes and a favorite recipe, as shown in figure 2.6.

Figure 2.6 Hello World example version 6 after parsing XML response

Problem

The server is sending structured data as XML. We need to parse this data on the client.

Solution

The first step in handling XML on the client side is to use the XHR's ability to parse the response into a structured XML document. The second step is to read (and potentially write) the parsed XML document using the W3C standard known as the Document Object Model (DOM). In JavaScript, we already have an implementation of the DOM for working with HTML web pages programmatically. The good news, then, is that we can leverage these existing skills to work with XML documents delivered by Ajax. Listing 2.9 shows the full code for version 6 of our Hello World application.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 2.9 hello6.html

```
<html>
<head>
<title>Hello Ajax version 6</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript' src='prototype.js'> </script>
<script type='text/javascript'>
window.onload = function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    new Ajax.Request(
      "hello6.jsp?name="+encodeURI(name),
      {
        method:"get",
        onComplete:function(xhr){
          var responseDoc = xhr.responseXML;      ❶ Reads response as XML
          update(responseDoc);
        }
      }
    );
  };
};

function update(doc){
  var personNode = doc
    .getElementsByTagName('person')[0];
  var initial = personNode
    .getAttribute('initial');
  var nameNode = personNode
    .getElementsByTagName('name')[0];       ❷ Extracts data using DOM
  var name = nameNode.firstChild.data;
  var likesNode = personNode
    .getElementsByTagName('likes')[0];
  var likesList = likesNode
    .getElementsByTagName('item');
  var likes = [];
  for (var i=0;i<likesList.length;i++){
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    var itemNode = likesList[i];
    likes[i] = itemNode
      .firstChild.data;
  }
  var recipeNode = personNode
    .getElementsByTagName('recipe')[0];
  var recipeNameNode = recipeNode
     .getElementsByTagName('name')[0];
  var recipeName = recipeNameNode.firstChild.data;
  var recipeSuggestNode = recipeNode
    .getElementsByTagName('serving-suggestion')[0];
  var recipeSuggest = recipeSuggestNode.firstChild.data;
  var ingredientsList = recipeNode
    .getElementsByTagName('ingredient');
  var ingredients = {};
  for(var i=0;i<ingredientsList.length;i++){
    var ingredientNode = ingredientsList[i];
    var qty = ingredientNode.getAttribute("qty");
    var iname = ingredientNode.firstChild.data;
    ingredients[iname] = qty;
  }

  $('helloTitle').innerHTML =
    "<h1>Hello, <b><i>"
    +name
    +"</i></b></h1>";
  var likesHTML = '<h5>'
    +initial
    +' likes...</h5><hr/>';
  for (var i=0;i<likes.length;i++){
    likesHTML += likes[i]+"<br/>";
  }
  $('likesList').innerHTML = likesHTML;
  var recipeHTML = "<h5>"
    +initial
    +"'s favorite recipe is "
    +recipeName
    +"</h5>";
  for (key in ingredients){
    recipeHTML += key+" : "
      +ingredients[key]
      +"<br/>";
  }
  recipeHTML+="<br/><i>"
    +recipeSuggest
    +"</i>";
  $('ingrList').innerHTML=recipeHTML;
}

</script>
</head>
```

❷ **Extracts data using DOM**

❸ **Assembles HTML**

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

The first step here is by far the easiest. We can retrieve the response as an XML document object simply by reading the `responseXML` property ❶ rather than `responseText`. We've then rewritten our `update()` function to accept the XML object. All we need to do now is read the individual data elements from the XML object ❷ and render the data as HTML content ❸.

In practice, neither of these steps is difficult, but they are rather lengthy. Using the DOM methods and properties such as `getElementsByTagName()`, `getAttribute()`, and `firstChild`, we can drill down to the data we want, but we need to do it step by step. These properties and methods are identical to the ones that we use when working with HTML documents, so we won't deal with them individually here. If you've used the DOM to manipulate HTML, everything should look familiar. If you haven't, then there is plenty of information on these methods online.

Once we have extracted the data that we need, then we simply assemble the HTML content necessary to update the UI.

We've already discussed the many scenarios under which it might make sense for the server to generate XML. To keep things simple in this example and avoid in-depth coverage of technologies that only apply to a single programming language, we've simply generated the XML by hand in our JSP. Listing 2.10 presents the JSP for the sake of completeness.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 2.10 hello6.jsp

```
<jsp:directive.page contentType="text/xml"/>
<%
String name=request.getParameter("name");
%>
<person initial="<%=name.substring(0,1).toUpperCase()%>">
 <name><![CDATA[<%=name%>]]></name>
 <likes>
  <item>JavaScript</item>
  <item>Skiing</item>
  <item>Apple Pie</item>
 </likes>
 <recipe>
  <name>apple pie</name>
  <ingredient qty="3kg">apples</ingredient>
  <ingredient qty="1kg">sugar</ingredient>
  <ingredient qty="2.4kg">pastry</ingredient>
  <serving-suggestion>
   <![CDATA[Best Eaten Outdoors! Mmm!]]>
  </serving-suggestion>
 </recipe>
</person>
```

Note that we've set the `contentType` of our response as `text/xml` in this case. We've been doing this throughout our examples, largely as a show of good habits. In this case, though, we have a strong practical reason for doing so. If we don't set the MIME type to some type of `xml` (either `text/xml` or `application/xml` will do), then the `responseXML` property of the XHR object won't be populated correctly.

The rest of the JSP is unremarkable. To a seasoned Java and XML coder, it might also look overly simplistic, with the XML being handwritten as text. A more robust solution would be to use a library like JDOM to generate the XML document programmatically, and we encourage the reader to do that in practice. However, we've left it simple here—maybe painfully simple—to show the intent of what we're doing without getting too deeply into Java-specific libraries. After all, our main aim in this book is to teach client-side techniques, and our choice of Java rather than PHP, Ruby, or .NET was essentially arbitrary.

So, getting back to the code, we've simply created a template of the XML document, most of which contains dummy data, and added in a few dynamic values along the way. Let's move on to evaluate our experience with this example.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

Our first encounter with XML and Ajax has been rather mixed. Initially, things looked pretty good, given the special support for XML baked into the XHR object. However, manually walking through the XML response using the DOM was rather lengthy and uninspiring. Experience of this sort of coding has been sufficient to put a lot of developers off XML in favor of JSON.

The DOM is a language-independent standard, with implementations in Java, PHP, C++, and .NET, as well as the JavaScript/web browser version that we're familiar with. When we look at the use of XML outside of the web browser, we find that the DOM is not very widely used and that other XML technologies exist that make working with XML much more palatable. Thankfully, these technologies are available within the browser too, and we'll see in the next section how we can use them to make Ajax and XML work together in a much happier way.

2.3.2 Better XML handling with XSLT and XPath

The XML techniques that we saw in the previous example represent the core functionality that is available free of charge via all implementations of the XHR object. Working directly with the DOM is not pleasant, especially if you're used to more modern XML-handling technologies in other languages. The most common of these tools are XPath queries and Extensible Stylesheet Language Transformations (XSLT) transforms.

XPath is a language for extracting data out of XML documents. In listing 2.9, we had to drill down through the document one node at a time. Using XPath, we can traverse many nodes in a single line. XSLT is an XML-based templating language that will allow us to generate any kind of content, such as, for instance, HTML, from our XML document more easily, and also separate out the logic from the presentation rather better than we've been doing so far. XSLT style sheets (as the templates are known—no relation to Cascading Style Sheets) use XPath internally to bind data to the presentation.

The good news is that XSLT transforms and XPath queries are available on many browsers, specifically on Firefox and Internet Explorer. Even better, these are native objects exposed to the JavaScript engine, so performance is good. Safari hasn't yet provided a native XSLT processor, so this isn't a good option if support for a (non-Firefox) Mac audience is important.

In the following example, we'll let Prototype have a well-earned rest, and use the Sarissa library to demonstrate simple cross-browser XSLT and XPath as a way of simplifying our XML-based Hello World example.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

Working with the DOM on our Ajax XML responses is slow and cumbersome. We want to use modern XML technologies to make it easy to develop with Ajax and XML.

Solution

Use XPath and XSLT to simplify things for you. Both Internet Explorer and Firefox support these technologies, but in quite different ways. As with most cross-browser incompatibilities, the best strategy is to use a third-party library to present a unified front to our code. For this example, we've chosen Sarissa (http://sarissa.sf.net), which provides cross-browser wrappers for many aspects of working with XML in the browser. Listing 2.11 shows the client-side code for our XSLT and XPath-powered app.

Listing 2.11 hello7.html

```
<html>
<head>
<title>Hello Ajax version 7</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript'
  src='sarissa.js'> </script>
<script type='text/javascript'
  src='sarissa_ieemu_xpath.js'> </script>
<script type='text/javascript'
  src='sarissa_dhtml.js'> </script>
<script type='text/javascript'>

var xslDoc=null;

window.onload=function(){

  xslDoc=Sarissa.getDomDocument();
  xslDoc.load("recipe.xsl");

  document.getElementById('helloBtn')
```

❶ Imports Sarissa libraries

❷ Loads XSL style sheet

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    .onclick = function(){
      var name = document.getElementById('helloTxt').value;
      var xhr = new XMLHttpRequest();
      xhr.open("GET",
        "hello7.jsp?name="
        +encodeURI(name),true);
      xhr.onreadystatechange = function(){
        if (xhr.readyState == 4){
          update(xhr.responseXML);
        }
      };
      xhr.send("");
    };
  };

  function update(doc){
    var initial = doc.selectSingleNode(
      '/person/@initial'
    ).value;
    var name = doc.selectSingleNode(
      '/person/name/text()'
    ).nodeValue;
    document.getElementById('helloTitle')
     .innerHTML = "<h1>Hello, <b><i>"
        +name+"</i></b></h1>";

    var likesList = doc
      .selectNodes('/person/likes/item');
    var likes = [];
    for (var i=0;i<likesList.length;i++){
      var itemNode = likesList[i];
      likes[i]=itemNode
        .firstChild.data;
    }
    var likesHTML='<h5>'
      +initial+' likes...</h5><hr/>';
    for (var i=0;i<likes.length;i++){
      likesHTML += likes[i]+"<br/>";
    }
    document.getElementById('likesList')
      .innerHTML = likesHTML;

    var personNode = doc.selectSingleNode('/person');

    var xsltproc = new XSLTProcessor();
    xsltproc.importStylesheet(xslDoc);
    Sarissa.updateContentFromNode(
      personNode,
      document.getElementById('ingrList'),
      xsltproc
    );
  }
```

❸ **Creates XHR object**

❹ **Assigns callback function**

❺ **Selects individual nodes**

❻ **Selects multiple nodes**

❼ **Invokes XSLT transform**

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div id='ingrList' class='sidebar'>
<h5>Ingredients</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

The first thing that we need to do is to import the Sarissa libraries ❶. As well as importing the core library, we import a support library that provides IE-style XPath under Firefox, and a helper library that offers some convenience methods for inserting XSLT-generated content into web pages.

Generating content using XSLT requires two XML documents from the server: the style sheet (that is, the template) and the data. We'll fetch the data on demand, as before, but can load the style sheet up front when we load the app ❷. We do this using a DomDocument object rather than the XHR. Once again, Sarissa provides us with a cross-browser wrapper.

To load the XML data, we will use an XHR object. Because we've put Prototype aside for this example, we need to create the XHR object by hand ❸ and assign the callback ❹. Nonetheless, the code is simpler than we saw in chapter 1, because we can access a native XHR object, even on Internet Explorer. Internally, Sarissa does a bit of object detection, and if no native XHR object can be found, it will create one for us that secretly creates an ActiveX control and uses it.

So, once we've got our XHR object, we can pass a DOM object to our `update()` function. This was where our troubles started when using the DOM. Using XPath, we can drill down through several layers of DOM node in a single line of code. For example, the XPath query

```
/person/name/text()
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

selects the internal text of a <name> tag nested directly under a <person> tag at the top of the document. XPath is too big a subject for us to tackle in depth here. We suggest http://zvon.org as a good starting place for newcomers to XPath and XSLT. The DOM Node methods `selectSingleNode()` ❺ and `selectNodes()` ❻ are normally only found in Internet Explorer, but the second Sarissa library that we loaded has provided implementations for Firefox/Mozilla. We're using XPath to extract the name data and the list of likes, and constructing the HTML content for those regions of the screen manually, as they're relatively straightforward. The recipe section is more complex, so we'll use that to showcase XSLT.

The final step is to perform the XSLT transform ❼. The XSLTProcessor object is native to Mozilla, and provided under IE by Sarissa. We pass it a reference to the style sheet, and then call a method `updateContentFromNode()`. This helper method, provided by the third Sarissa library that we loaded, will pass the data (i.e., `personNode`) through the XSLT processor and write the resulting HTML into the specified DOM node (i.e., `ingrList`).

To make this work, of course, we also need to provide an XSL style sheet. That's shown in listing 2.12.

Listing 2.12 recipe.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
  Transform">
<xsl:output method="xml"/>

 <xsl:template match="/person">
  <div>
  <h5><xsl:value-of select='@initial'/>'s
   favorite recipe is
  <xsl:value-of select='recipe/name'/></h5>
  <p><xsl:apply-templates select="recipe/ingredient" /></p>
  <p><i><xsl:value-of select='recipe/serving-suggestion'/></i></p>
  </div>
 </xsl:template>

 <xsl:template match="ingredient">
  <xsl:value-of select='@qty'/> : <xsl:value-of select='.'/><br/>
 </xsl:template>

</xsl:stylesheet>
```

Our XSL style sheet is quite straightforward. It's a mixture of ordinary XHTML markup, and special tags prefixed with `xsl`, indicating the XSL namespace. These

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

are treated as processing instructions. `<xsl:template>` tags specify chunks of content that will be output when a node matching the XPath query in the match attribute is encountered. `<xsl:value-of>` prints out data from the matched nodes, again using XPath expressions. The `<xsl:apply-templates>` tag routes nodes to other template tags for further processing. In this case, each ingredient node will be passed to the second template from the first, generating a list.

Again, we don't have space for a full exposition of XSLT style sheet rules here. If you wish to know more, we recommend you visit http://zvon.org.

Finally, let's turn briefly to the server side. The JSP used in this example is identical to that from the previous example, as presented in listing 2.10. The only changes that we've introduced have been in the client code.

Discussion

Using XSLT and XPath has certainly simplified our client-side XML-handling code. In a more complex application, these technologies will scale more easily than the DOM in terms of coding effort. We recommend that anyone considering using Ajax with XML investigate these technologies.

In our section on JSON, we discussed the notion of round-tripping structured data between the client and the server. Sarissa promotes this approach, using XML as the interchange format, as it also supports cross-browser serialization of XML objects. As we stated earlier, almost any server-side technology will provide support for serializing and deserializing XML, too. We won't explore a full example here, but the principle is similar to the JSON case. When using JSON with Java, we noted that a fair amount of manual work was required to construct the JSON response because of the mismatch between loosely typed JSON and strictly typed Java. The same issues exist when converting between Java and XML, but the problem space is better understood, and out-of-the-box solutions such as Castor and Apache XMLBeans are available.

We've presented Sarissa as a one-stop shop for these technologies. It isn't the only game in town. If you only want XPath queries, then the mozXPath.js library (http://km0ti0n.blunted.co.uk/mozXPath.xap) provides a lightweight alternative, with support for the Opera browser as well. And if you like the look of XSLT but need it to work on Safari, then you can try Google's AJAXSLT, a 100 percent JavaScript XSLT engine (http://goog-ajaxslt.sf.net). Be warned, though, that AJAXSLT is slow compared to the native engines in IE and Mozilla and won't support the full XSL namespace, so you'll need to write your style sheets with the limitations of the library in mind and keep them reasonably small.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We're nearly done with our review of Ajax technologies. In the final section, we'll explore another Internet technology that makes use of XML, SOAP web services, and see how Ajax can interface with that.

2.4 Using Ajax with web services

In this section, we will see how to call web services running on a remote server over SOAP. After all, what is a web service but XML data being passed back and forth? The XHR object is ideally suited for such a task and makes invoking remote methods over SOAP less of a daunting task than it may seem.

Internet Explorer and the Mozilla versions of browsers all have native objects that can be used to invoke web services. Sadly, these objects are not portable between browsers; the developer is left to write a custom framework that can choose the proper objects to invoke. Microsoft maintains several pages dedicated to its version of browser-side SOAP at http://msdn.microsoft.com/workshop/author/webservice/overview.asp. Microsoft's implementation is based on both JavaScript and VBScript. Mozilla explains their version at www.mozilla.org/projects/webservices/; more information can also be found at http://developer.mozilla.org/en/docs/SOAP_in_Gecko-based_Browsers. Their version of browser-side SOAP is accessible through native objects that can be constructed on the browser side.

Fortunately, there is another way. Instead of writing a high-level API that can make use of either Internet Explorer or Mozilla objects, we can create our own library that uses XMLHttpRequest to exchange XML, and that can parse and generate the SOAP messages. Such a library would also allow us to run our code on browsers that do not supply either the Microsoft or Mozilla SOAP APIs but that do have the XHR object. The kind people at IBM have created just such a library and have named it ws-wsajax. It can be found at www.ibm.com/developerworks/webservices/library/ws-wsajax/. We will be using this library for the remainder of this section.

We've simplified the UI for this example, removing the recipe section. Passing in the name will return a map with three entries: the name, the initial, and the list of likes. Figure 2.7 shows the UI for this example.

This section assumes some familiarity with SOAP and SOAP-RPC. Once again, there are several books available, as well as many good tutorials online, that cover this topic in depth.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Figure 2.7 Hello World version 8. We've simplified the presentation here, removing the recipe element from the UI.

Problem

You need to perform SOAP-RPC from a web browser. You need to display the resulting SOAP response as HTML.

Solution

In this section, we will write a small client using IBM's SOAP toolkit to access our own Hello World SOAP service, written using Apache's Axis framework (http://ws.apache.org/axis/). Let's begin by defining our web service. Axis makes it very easy to prototype web services by writing Java classes in files with a special filename extension: `.jws`. Like JSPs, `.jws` files will be compiled on demand by a special servlet, in this case the AxisServlet, and, while not robust enough for production use, serve the purposes of our simple demonstration admirably. Listing 2.13 shows a simple `.jws` file for our Hello World service.

Listing 2.13 HelloWorld.jws

```
import java.util.Map;
import java.util.HashMap;

/**
 * class to list headers sent in request as a string array
 */
public class HelloWorld {

    public Map getInfo(String name) {
      String initial=name.substring(0,1).toUpperCase();
      String[] likes=new String[]
  { "JavaScript", "Skiing", "Apple Pie" };
      Map result=new HashMap();
      result.put("name",name);
      result.put("initial",initial);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      result.put("likes",likes);
      return result;
    }

}
```

The class contains a single method, which will be mapped to a SOAP-RPC function. The function takes one argument, of type String, and returns an associative array (referred to in Java as a Map).

Pointing our browser at `HelloWorld.jws` will return a Web Service Description Language (WSDL) file, which the SOAP client, such as the IBM library, can interrogate in order to build up client-side stubs, allowing us to call the service. Listing 2.14 shows the WSDL generated by this class.

Listing 2.14 WSDL for HelloWorld.jws

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:apachesoap="http://xml.apache.org/xml-soap"
  xmlns:impl="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:intf="http://localhost:8080/AiP2/HelloWorld.jws"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema
    targetNamespace="http://xml.apache.org/xml-soap"
    xmlns="http://www.w3.org/2001/XMLSchema">
   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
   <complexType name="mapItem">
    <sequence>
     <element name="key" nillable="true" type="xsd:anyType"/>
     <element name="value" nillable="true" type="xsd:anyType"/>
  </sequence>
  </complexType>
   <complexType name="Map">
    <sequence>
     <element maxOccurs="unbounded" minOccurs="0"
       name="item" type="apachesoap:mapItem"/>
    </sequence>
   </complexType>
  </schema>
 </wsdl:types>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    <wsdl:message name="getInfoResponse">
      <wsdl:part name="getInfoReturn" type="apachesoap:Map"/>
    </wsdl:message>
    <wsdl:message name="getInfoRequest">
      <wsdl:part name="name" type="xsd:string"/>
   </wsdl:message>
   <wsdl:portType name="HelloWorld">
      <wsdl:operation name="getInfo" parameterOrder="name">
         <wsdl:input message="impl:getInfoRequest"
            name="getInfoRequest"/>
         <wsdl:output message="impl:getInfoResponse"
            name="getInfoResponse"/>
      </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSoapBinding" type="impl:HelloWorld">
      <wsdlsoap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="getInfo">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="getInfoRequest">
            <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://DefaultNamespace" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="getInfoResponse">
            <wsdlsoap:body
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="http://localhost:8080/AiP2/HelloWorld.jws"
              use="encoded"/>
         </wsdl:output>
     </wsdl:operation>
   </wsdl:binding>
    <wsdl:service name="HelloWorldService">
      <wsdl:port binding="impl:HelloWorldSoapBinding"
        name="HelloWorld">
         <wsdlsoap:address
           location="http://localhost:8080/AiP2/HelloWorld.jws"/>
      </wsdl:port>
    </wsdl:service>
  </wsdl:definitions>
```

The WSDL includes details on the argument types and return types of each RPC call, bindings to the functions, and other details needed by the client and server to specify the nature of the interchange. Fortunately, the WSDL is generated for us automatically by Axis and is consumed by the IBM toolkit, so we don't need to understand every line in it.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Let's turn now to our client-side code. Listing 2.15 shows the full listing for version 8 of our Hello World app.

Listing 2.15 hello8.html

```
<html>
<head>
<title>Hello Ajax version 8</title>
<style type='text/css'>
* { font-family: Tahoma, Arial, sans-serif; }
#helloTitle{ color: #48f; }
.sidebar{
  background-color: #adf;
  color: navy;
  border: solid blue 1px;
  width: 180px;
  height: 200px;
  padding: 2px;
  margin: 3px;
  float: left;
}
</style>
<script type='text/javascript'
  src='prototype_v131.js'> </script>                    ❶ Imports Prototype
<script type='text/javascript' src='ws.js'> </script>    ❷ Imports IBM WS library
<script type='text/javascript'>

window.onload=function(){
  $('helloBtn').onclick = function(){
    var name=$('helloTxt').value;
    var wsNamespace = '../axis/HelloWorld.jws';           ❸ Creates client
    var wsCall = new WS.Call(wsNamespace);                   from WSDL
    var rpcFunction = new
      WS.QName('getInfo',wsNamespace);                    ❹ References RPC function
    wsCall.invoke_rpc(
      rpcFunction,
      [{name:'name',value:name}],                         ❺ Passes RPC arguments
      null,
      function(call,envelope){                            ❻ Defines callback
        var soapBody = envelope.get_body();
        var soapMap = soapBody
          .get_all_children()[1].asElement();
        var itemNodes = soapMap
          .getElementsByTagName('item');
        var initial = "";
        var likes = [];
        for (var i=0;i<itemNodes.length;i++){
          var itemNode = itemNodes[i];
          var key = itemNode
            .getElementsByTagName('key')[0]
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            .firstChild.data;
          if (key == 'initial'){
            initial = itemNode
              .getElementsByTagName('value')[0]
              .firstChild.data;
          }else if (key == 'likes'){
            var likeNodes = itemNode
              .getElementsByTagName('value')[0]
              .getElementsByTagName('value');
            for (var j=0;j<likeNodes.length;j++){
              likes[likes.length] = likeNodes[j]
                .firstChild.data;
            }
          }
        }
        update(initial,likes);
      }
    );
  };
};

function update(initial,likes){    <-- 7 Updates UI
  var content = "<h5>"+initial
    +" likes...</h5><hr/>";
  for (var i=0;i<likes.length;i++){
    content += likes[i]+"<br/>";
  }
  $('likesList').innerHTML = content;
}

</script>
</head>
<body>

<div id='likesList' class='sidebar'>
<h5>Likes</h5><hr/>
</div>
<div>
<div id='helloTitle'>
<h1>Hello, stranger</h1>
</div>
<p>Please introduce yourself by entering your name
   in the box below</p>
<input type='text' size='24' id='helloTxt'></input>
 
<button id='helloBtn'>Submit</button>
</div>
</body>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

There's a lot going on here, so let's take it line by line. First, we need to import the IBM library ❷. Because this library is built on top of Prototype, we include that too. It relies on an older version of Prototype (v1.3.1), so we've renamed it to avoid confusion with the rest of our examples ❶.

To consume the Web Service, the first thing that we need to do is reference the WSDL and feed it to a `WS.Call` object ❸. We then extract a reference to the specific function, as a `WS.QName` object ❹. We can call this object, providing the input parameters as a JavaScript object (which we've defined inline here using JSON) ❺, and a callback function to parse the response ❻. Parsing the response requires a lot of node traversal. We're working with SOAP nodes rather than DOM nodes here, but the SOAP nodes can be converted to DOM nodes at any point. We've omitted any use of XPath here to keep the example simple, but wading through a larger SOAP response would certainly merit investigating use of XPath. Once we have extracted the data from the response, we pass it to our `update()` function ❼, as usual. Again, we've opted for simplicity here, but there's nothing to stop you from using XSLT transforms on the SOAP response once you've got ahold of it.

Discussion

We've shown that it's possible to use SOAP with Ajax, provided of course that the SOAP service is coming from the same server as the Ajax client, and therefore honoring the browser's same-origin security restrictions. If your back-end system already generates SOAP, then this is a valid way of reusing existing resources. However, we'd be tempted to say that SOAP, as an architecture for a green-field development, is unnecessarily complex if interoperability with external entities is not also a requirement.

The IBM SOAP toolkit made it very easy to call the service, but somewhat less easy to parse the response. SOAP-RPC responses typically involve several namespaces and are complex to decode. Document/literal-style SOAP bindings generally provide simpler responses, which might be a better fit for this toolkit in production.

As always, caveat programmer. If you need a quick solution, SOAP may not be the way to go. However, if you are creating a large application that you foresee will require many updates and extensions, as well as integration with many aspects of your organization, and you have the time and skills to do it, browser-side SOAP may benefit you.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

2.5 Summary

By the end of chapter 1, we'd figured out how to make an Ajax request, and looked at ways of simplifying the process by using third-party libraries. We've covered a lot of ground since then and shifted our focus from simply being able to make a request, to looking at how we want to structure the conversation between client and server over the lifetime of the application.

We've looked at several techniques in this chapter and evaluated the strengths and weaknesses of each. We began by looking at generating JavaScript code on the server and saw the benefits of writing generated code against a high-level API in order to prevent excessive tangling between the client and server codebases.

We moved on from there to look at ways of passing structured data between the client and server, starting with JSON and then continuing on to XML. In each case, we began by simply looking at how to parse the data when it arrived from the server, and then moved on to consider the full round-trip of data between client and server. By round-tripping the data, and having library code to serialize and deserialize at both ends, we can free ourselves up to write business code rather than low-level plumbing.

In contrast to JSON and XML, JSON has a closer affinity with the client side. We struggled with our client-side XML initially but made significant advances when we picked up XPath and XSLT. There is no clear winner between the two technologies, and the decision remains a matter of personal taste, and depends on whether you are integrating with legacy systems that naturally fit better with either JSON or XML.

In the next chapter, we'll look at JavaScript as the programmatic glue that binds the entire Ajax app together. We'll discuss recent advances in thinking about JavaScript, and how they can help you to write better-structured code for your Ajax app. We'll conclude with a discussion of some of the popular Ajax frameworks.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3 Object-oriented JavaScript and Prototype

This chapter covers

- Working with core JavaScript types
- Writing effective object-oriented JavaScript
- Using the Prototype library

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If you are like most web developers, you probably got your start with JavaScript by writing small and simple scripts on fairly basic web pages. Perhaps it was to create image rollovers, or to perform simple form validation on entry fields prior to form submission.

But if you're reading this book, your web application skills have probably progressed far beyond those humble beginnings—and so has the nature of the script being written for the pages of your web applications.

Given the nature of modern web applications, the amount and complexity of the code that goes into them (whether on the server or the client side of the equation) is steadily growing. And once you add Ajax to the mix, the complexity and sheer amount of client-side code gets a significant bump for even modest applications.

This creates a compelling need for JavaScript code to "grow up" and be treated with the same level of care and respect previously reserved for code written in server-side languages such as Java, C++, and C#. Organizing our client-side code with the same care as its server-side counterparts not only helps us maintain our own sanity with regard to its creation, it also facilitates readability, reuse, testing, extensibility, and the maintainability of the code.

One prevalent methodology used to organize the code that is common to all the server-side languages we've mentioned is *object orientation*. While JavaScript *is* an object-oriented (OO) language, it does lack some of the OO concepts and capabilities that other object-oriented languages possess. But that doesn't mean we cannot benefit from the lessons and concepts that are so easily available to such languages. JavaScript may not have all the OO bells and whistles that Java or C++ can boast about, but it has some unique features of its own that those languages lack. In this chapter you'll learn exactly what those features are, and how they can be exploited to use object-oriented concepts and techniques to bring order to your client-side JavaScript.

The pedantic might argue that what we discuss in this chapter is not *truly* object-oriented code but rather object-orientation-*influenced* code. Whatever. We'll call it object-oriented JavaScript and we'll all know what we really mean.

First, we'll take a look at the unique aspects of JavaScript that make such code possible, and learn how to use it to our advantage to create better-organized code using OO techniques and concepts. Then we'll introduce a freely available JavaScript library named Prototype and see how it can help us write better JavaScript, with a focus on better *object-oriented* JavaScript.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3.1 Object-oriented JavaScript

It shouldn't come as a surprise to anyone that the concept of an *object* is the core of any object-oriented language. And so it is with JavaScript, but in ways that might seem rather mysterious or just downright strange to anyone familiar with the more traditional OO languages such as Java or C++.

The key to understanding how to best make use of JavaScript's OO features lies not only with a good understanding of how the JavaScript object works, but also in a thorough understanding of how JavaScript functions operate. Indeed, it is the rather unique and interesting fashion in which functions are implemented that is crucial to grasping how to best make use of JavaScript's features to rein in complex code.

As such, we'll explore both the concept of the JavaScript object as well as JavaScript functions in this section. After an overview of the features and operation of each, we'll show how these concepts can be combined to form the basis of what we are calling object-oriented JavaScript.

Let's start by taking a look at the fundamental concepts behind the JavaScript object.

3.1.1 Object fundamentals

The first step in wrapping your head around the concept of a JavaScript object is to rid your mind of any preconceived notions about how the Object class—usually the basic unit from which all other objects are built—is implemented in other languages. Even though in JavaScript the Object class *is* the fundamental unit from which all other objects are built, the resemblance doesn't go much further than that.

Perhaps the best way to think of objects in JavaScript is as an unordered collection of key-value pairs, very similar to the concept of a Map or Dictionary in other languages. This pair is called a *property*, and consists of a *key* or name that identifies the property, and a *value* that the property possesses.

You might think that properties are similar to member variables of other languages, but because JavaScript is not a declarative language in which the members of a class need to be declared, properties of objects are created on the fly by simply assigning a value to them. Moreover, the data type of these properties is dynamic rather than predeclared. A property takes on the data type of any value assigned to it, and it can even change data types over its lifetime.

Let's take a look at a simple example. Suppose we wanted to keep track of our collection of music CDs. Each object that represents a CD will have properties that

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

record its title, the artist or band, and the number of the shelf or drawer that we keep it in. Did we mention that we have a *lot* of these?

There are, of course, many other attributes of a CD that we could create properties for. But for illustrative purposes, these three will suffice. Setting up such an object might look like the code shown in listing 3.1.

Listing 3.1 Creating and populating a CD object

```
var aCD = new Object();
aCD.title = 'The Lovin\' Spoonful Greatest Hits';
aCD.artist = 'The Lovin\' Spoonful';
aCD.location = 3;
```

In listing 3.1, the first thing that we did was create an instance of the Object class by invoking the `new` operator on the Object constructor. Although this seems almost a trivial task, there are some important nuances that it's crucial to understand regarding the `new` operator.

Up to now, you may not have thought of `new` as an operator, but that's exactly what it is. Its operand is a function that it construes to be the *constructor* for the object that `new` creates.

We'll get into constructors a bit more after we've examined functions in greater detail, but for now suffice it to say that the object created by `new` is created with the help of the constructor function specified as its operand, which in the case of the predefined JavaScript Object class generates a blank object instance; that is, one with no properties.

In our example, the blank object is assigned to a variable named `aCD`, ready for us to assign properties, which we do over the next three lines.

Note that we didn't need to predeclare that our object could accept these properties, as we would have needed to do in a declarative language. In JavaScript, the act of assigning values to the properties causes them to come into existence.

Properties are most often referenced using the "dot" (period) operator as shown in our example, but may also be referenced using the more general "property accessor" operator. Using this general notation, the assignment to the `location` property could have been

```
aCD['location'] = 3;
```

This is completely equivalent to the dot notation of `aCD.location = 3`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The dot notation can be used whenever the property name adheres to the format for an identifier. If the property name violates that form—for example, say it contains a space character—the bracket notation must be used:

```
ACD['the location of the CD'] = 3;
```

Generally, in the interest of readability, object properties are usually given names that follow the format used for identifiers.

So now we have an instance of an object that contains three properties describing a CD. This is all well and good, but if you think about it, it's not all that scalable. It took four lines to create and populate the object. If we were to enter many CDs—remember, we warned you this was a large collection—we'd need a lot of lines of code.

Even if we weren't concerned about the sheer amount of code, the possibility of error is high. Since we are explicitly setting the property names on each instance of creating a CD object, a typo in any one set of assignments throws a hard-to-find monkey wrench into the works. Remember that we can create any property simply by assigning it. Let's imagine that we were to accidentally type

```
anotherCD.lcoation = 213;
```

No errors would occur at the time of the assignment, but at some point later on down the line we'd wonder why that instance was missing its `location` property.

What we'd really like to do is to create a *constructor* for the CD instances that handles all of this consistently and internally, employing the object-oriented concept of *encapsulation*—perhaps something along the lines of

```
var aCD = new CD('The Very Best of the Rascals', 'The Rascals', 6);
```

That way, a single block of code (the constructor) would take care of making the property assignments, thus removing the possibility of typos in repeated blocks of code.

But before we can discuss the use of functions as constructors, we need to understand how functions themselves operate in JavaScript. Let's take a look.

3.1.2 Functions are first class

While JavaScript functions may at first seem to share many similarities with methods of traditional OO languages, some vast differences become apparent as soon as we start scratching the surface a bit.

A large part of these differences lies in the fact that JavaScript functions are treated as "first-class" objects within the language. That doesn't mean they get

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

wider seats when flying, but rather that they are on equal footing, and share characteristics with, the other object types in the language. They can be created on the fly, created via anonymous literals, referenced by variables, passed as parameters, returned as the results of other functions, and in general, treated like any other data value.

Let's start by taking a look at how functions can be declared and invoked.

Declaring and calling functions

Before a function can be invoked, it must have come into existence via its declaration. At its simplest, a function could be declared using syntax that is probably quite familiar:

```
function doSomething(value) {
  alert("I'm doing something with " + value);
}
```

But functions don't need to be named. They can be created via a *function literal*, also sometimes called an *anonymous function*:

```
function(value) {
  alert("I'm doing something with " + value);
}
```

That's interesting, but (at least in this example) not all that useful since we have no way of actually calling such a function. How useful is a function that can never be invoked?

But remember that functions, as first-class objects, can be assigned to variables. When such a reference exists, the function can be invoked through that reference. This is true not only of function references in variables, but also of references passed as function parameters, and of functions stored as properties of an Object instance.

This opens up a lot of interesting possibilities. Consider the following:

```
var doSomething = function(value) {
  alert("I'm doing something with " + value);
}
```

Now, the code fragment `doSomething('some value')` can be used just as if we had declared the function with a name.

Aside from being an interesting alternative syntax, this ability to invoke a function given a reference to it comes in very handy in other scenarios. Consider this code:

```
function saySomething(text) { alert('value: ' + text); }

function doSomething(value,onComplete) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    // does something with value
    onComplete(value);
  }

  doSomething(213,saySomething);
```

The first function takes a value and constructs an alert using that value, while the second performs some function on a passed value. The interesting thing about the second function is that, when the processing is complete, it has allowed its caller to customize whatever notification is to occur by allowing a reference to a *callback function* to be passed to it as the parameter `onComplete`.

The callback function is passed as a reference to the processing function and is invoked via that reference. This allows the *caller* of the function, rather than the function itself, to determine what occurs when the processing is complete.

"So what?" you might be thinking. "Why would I want to go through all that when I can just do whatever I want after the processing function call returns?"

That might be true in a synchronous fragment of code, but this ability to call back to functions by reference becomes much more interesting and compelling when we start to throw asynchronous scenarios, such as input events and Ajax, into the mix.

That aside, the main point of this code example was to show how function references could be used as parameters to other functions. Function references can also be used as *property values* for Object instances. Consider:

```
var o = new Object();
o.doSomething = function() { alert('Yo!'); }
```

In this case, the function could be invoked with

```
o.doSomething();
```

When assigned as a property of an Object instance, the function is termed a *method* of that object—a concept that is not as superficial as you might at first think. Storing a reference to a function in an object's property serves as more than just a place to store a reference; it also creates an association between the method and the object within which it is referenced.

Which brings us to the concept of the *function context*. Let's see what that's all about.

Understanding function contexts

All functions execute in the *context* of a JavaScript object, even if you never realized that. That object, termed the *function context*, is available to the body of the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

function via the reserved word `this`, which you might be familiar with from languages such as C++ and Java.

When a function is invoked through a reference stored as the property of an object (making it a method of that object), the context object referenced by `this` in that function is the containing object. Even named functions not stored as object methods have a context object, consisting of the window object for the page.

It's important to note that the function context is an attribute of a function *invocation*, not of the function itself. Let's consider the code in listing 3.2.

Listing 3.2 Investigating function contexts

```
function xyz() {                                  <-1 Creates named function
  alert(this.handle);
}
                                                     2 Creates and assigns
var o = new Object();                          <-    function to object
o.methodXyz = xyz;

window.handle = "I'm the window";               <-3 Creates handle property
o.handle = "I'm o";

xyz();                 <-4 Invokes function twice
o.methodXyz();
```

In this code fragment we first create a named function, `xyz` (1), that issues an alert containing the value of the `handle` property of whatever object is referenced by `this`—in other words, the `handle` property of the function context object. Note that `handle` is not a built-in property of any object; it's a property that we'll be creating as a way to easily identify individual objects.

Next, an object is created and assigned to `o` (2). A property named `methodXyz` is created on that object that stores a reference to the `xyz` function. Then, a property named `handle` is created on the window object for the page, as well as on the object referenced by `o` (3). This will allow us to easily identify which of these objects is being referenced at any time.

We then invoke the function twice (4): once directly via the `xyz` name, and once via the `methodXyz` property of object `o`. Executing this code results in the display of two alerts as shown in figure 3.1.

These alerts, displayed one after the other, clearly demonstrate that even though the same function is being called in each case, the function context for the function is determined for each function *invocation* as a result of the manner in which the function is called.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Figure 3.1
Same function, different contexts!

The fact that the context for a function, when referenced via an object property as a method of that object, is the object itself is essentially what makes object-oriented JavaScript possible.

But sometimes, the JavaScript interpreter can supplant what we might normally expect to be the context object. What's up with that?

When a stranger holds the leash

We saw in the previous section that the context object for a function is determined by how the function is called. When the function is a method of an object, and is invoked through that object, the object is the function's context.

As we'll see in many of our examples, it is sometimes convenient for us to use class methods as event handlers—for example, as an `onclick` handler for a button. And that's when things get thrown into a bit of a loop. You see, when a function is invoked as a handler as a result of an event, the function context is set to be the element that triggered the event even if the function is already a method of another object.

Head spinning yet?

Sorry about that. Let's try a somewhat specious but hopefully helpful analogy.

Let's say that you have a pet iguana, which represents a function. Iguanas, not being the brightest of nature's creatures, need a way to refer to their owner, you, representing the function's context object. This connection is a leash, representing the `this` variable.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When the iguana needs to reference you, say for food, it uses the leash as a guide and is rewarded with a handful of healthy dandelion greens. But let's say the iguana (function) is invoked as a handler. In this case, the context object is changed to the event-initiating object—in other words, the leash is handed to a stranger. When the hungry iguana follows the leash to the interloper, he doesn't get fed. (After all, would *you* feed a strange iguana?)

That's a problem that we'll need to deal with in our examples. One means that we'll explore is setting properties on the element so that we can get back to the "right" object given a reference to the element. But later in this chapter we'll also see a clever means to force the JavaScript interpreter to bow to our wills with regard to a handler's context.

But before we get there, it *is* possible for us to control what object is to serve as the context of a function invocation when we're the ones in the driver's seat. This won't help us out in the handler case (as we're not the ones who make the call in that scenario), but when we are in control, let's check out how we can explicitly specify what object is to serve as the function context.

Setting the function context

Each function (as an instance of the JavaScript built-in class Function) has a method (remember since functions are first-class objects, they can possess properties and methods just like other objects) named `call()`.

When a function is invoked "normally," the function's context object is determined by the interpreter, as we previously discussed.

Consider the following code snippet:

```
function whatsYourName() {
  alert(this.name);
}

var o = { name: 'Felix the Cat' };

whatsYourName();
whatsYourName.call(o);
```

Executing this code results in the alerts that we see in figure 3.2.

In this code fragment, the `whatsYourName()` function issues an alert with the `name` property of whatever object is serving as its function context.

When the function is invoked directly, the alert displays the name property of the window since that object is supplied as the context object for that function invocation. The displayed string may be something along the lines of

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Figure 3.2 We have control over the function context should we wish it.

"file://localhost" or some such, depending on your browser and how you loaded the page into it.

We also defined an object `o` (using JSON notation) that possesses a `name` attribute. When the function is invoked using the `call()` method, supplying the `o` object as the parameter, it should be clear that `o` has been used as the function invocation's context.

Before we return to our discussion of objects with this newfound knowledge under our belts, there's another concept with regard to functions that we need to understand: the concept of a *closure*. Let's see what that's all about.

Closing in on closures

Closures are a concept that you might not have run into if you are coming here from the Java or C++ world, as there is no corresponding concept in these languages. It can be a difficult concept to grasp, so let's start with some sample code right off the bat, as shown in listing 3.3.

Listing 3.3 Creating a closure

```
var o = new Object();
o.setup = function() {
  var someText = 'This is some text';
  this.doSomething = function() {
    alert(someText);
  };
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
o.setup();
o.doSomething();
```

In the code snippet in listing 3.3, we create a new object assigned to `o`, and then create a `setup()` function to initialize it. (This is something better accomplished via a constructor, but we'll get to that before too much longer.)

In `setup()`, we create a local variable named `someText` and give it a string value. We then set up a method named `doSomething()` that simply emits an alert message displaying the value of that variable. We call the `setup()` function to initialize the object, and then call the `doSomething()` method.

On inspecting the code for that method, we detect a problem: the code references the `someText` variable, which was local to the block in which the function was created, but according to JavaScript rules, that variable went out of scope as soon as that block terminated.

Therefore, we'd expect that when the `doSomething()` function is called later—at the top level and clearly outside of the scope of the block that defined `someText`—an undefined reference would occur. But upon execution, we see an alert, as shown in figure 3.3.

Odd. The string was emitted in the alert. How can that be? Does this make us want to question what we thought we knew about JavaScript scoping rules?

The `someText` variable is indeed out of scope when we make the call to `o.doSomething()`. If we were to add the statement `alert(someText);` right after that call, we'd find that the JavaScript interpreter would indeed issue a "someText is not defined" error. So how did the method work?

What has happened is that when the JavaScript interpreter creates a function (as an instance of an object of type Function), it creates a *closure* for that function that is composed not only of the function itself but *also* of the environment that is in scope at the time that the function is created.

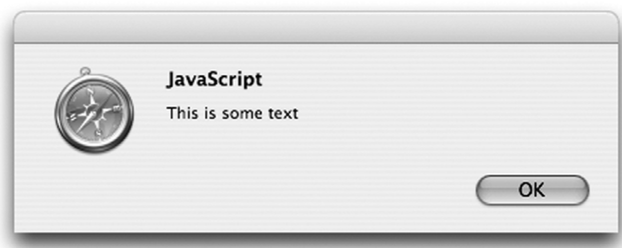

Figure 3.3 What hat did this get pulled out of?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

So anything that is in scope when a function is declared is *also available to the function when it is invoked*.

This is as powerful a concept as it is a confusing one and should be used with caution. You can create some really terrible code by using this technique thoughtlessly, but later on we'll see how closures will help make some elegant code in an object-oriented fashion.

Another important thing to note: the function context (the `this` pointer) of the executing function is never included in a closure when one is created. This is also something we'll see the implications of when we get deeper into defining JavaScript classes.

Finally, as promised, now that we know a little more about functions, we're ready to look at creating our own JavaScript classes in more detail, starting with constructors.

3.1.3 Object constructors and methods

Now that we know what a JavaScript object is, and have a better understanding of JavaScript functions, we're ready to see how we can use objects and functions to create well-organized JavaScript objects of our own via classes.

To start with, let's take a look at the `new` operator and how it operates on a function to transform that function into a constructor for an object.

Defining constructors

Consider the following code:

```
function Something(p1,p2,p3) {
  this.param1 = p1;
  this.param2 = p2;
  this.param3 = p3;
}
```

It's a pretty straightforward function that takes its parameters and stores them as properties on the current context object. Those familiar with object-oriented programming in Java or C++ will readily recognize this as following the usual pattern for constructors. But is it a constructor?

The answer is: yes and no—or perhaps more accurately, *it depends*.

Let's imagine that, after we have defined this function, we call it as follows:

```
Something(1,2,3);
```

What happens?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When the function executes, it creates three properties on its context object. In this case, because the function is simply being called from top-level code, that context object is the window object of the current page.

Well, that's not very interesting, is it?

But something much *more* interesting happens when, rather than calling the function directly, we apply the `new` operator to the function, as in

```
new Something(1,2,3);
```

The `new` operator creates a new, empty Object instance and then invokes the function supplied as its operand with that newly created object as the function invocation's context. The result is that when the function executes, the `this` pointer references the new object, thus turning the function into a constructor for that object.

That's more like it! Now, when `Something()` executes, it will create the three properties on the new instance of an object just as a constructor should.

That's great. We now know how to declare constructors and use them to initialize newly created objects. But aside from encapsulating the initialization of an object into a tidy package, what have we really gained?

Not to dismiss the advantages that such encapsulation gives us—review the example of listing 3.1—but in order for our object to be really useful, it needs more than just a constructor; it needs methods to act upon it in an object-oriented fashion.

We've previously seen how we can create methods by assigning functions to object properties, so let's examine how we can use that mechanism to further define our objects.

Adding methods

Let's take our CD example from earlier in the chapter and define a constructor for it, as shown in listing 3.4.

Listing 3.4 Constructor for the CD object

```
function CD(title,artist,location) {
  this.title = title;
  this.artist = artist;
  this.location = location;
}
```

Let's say that we wanted to add a method that would tell us where the CD is located. Remember that in order for a function to be a method of an object, it

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

must be referenced through a property of the object so that when invoked, its function context will be that object instance. We could do this by

```
var aCD = new CD('Afterburner','ZZ Top',17);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
};
```

But that won't do at all! If we had more than one CD (and why would we be doing this if that weren't the case?), then we'd need to perform this for each and every instance, as shown here (with the addition of an array to hold multiple instances):

```
var myCDs = new Array();
var aCD = new CD('Afterburner','ZZ Top',17);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
myCDs.push( aCD );
aCD = new CD('Mirage','Fleetwood Mac',7);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
myCDs.push( aCD );
aCD = new CD(''Please,'Pet Shop Boys',23);
aCD.whereIsIt = function() {
  alert( 'The CD is on shelf ' + this.location);
}
```

How silly, not to mention messy, would that be?

We could factor out the common code into a standalone function that we could assign as a property of each instance, but while that might reduce the propensity for error, that wouldn't make this any more scalable. Instead, as with the code to set up properties that we factored into the constructor, we want to encapsulate the creation of the methods into a tidy little package.

One way might be to place the creation of the methods within the constructor itself. The result could be as shown in listing 3.5.

Listing 3.5 Constructor for the CD object

```
function CD(title,artist,location) {
  this.title = title;
  this.artist = artist;
  this.location = location;
  this.whereIsIt = function() {
    alert( 'The CD is on shelf ' + this.location);
  };
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

That achieves our goal of encapsulating the creation of the method(s), but has a drawback in that each and every constructed object will possess a copy of this function. That's rather wasteful as, unlike properties such as `title`, which will be unique for each instance, each instance of CD will have an identical `whereIsIt()` function.

What would be ideal would be if there were a way to have *all* instances of the class reference a *single* instance of the function as their method. And that's where the `prototype` mechanism of JavaScript constructors comes in.

Let's find out what that's all about.

Defining prototypes

When an instance of an object is created as a result of the `new` operator, that object is initially empty; that is, it contains no properties. However, there's more than you might think going on behind the scenes.

When we make a reference to a property of an object, the JavaScript interpreter looks in the object instance for a property with the referenced name. That is exactly as we might expect.

But what you might not have known is that it doesn't stop there. If no such property is found on the object itself, the interpreter looks to the constructor for the object in a last-ditch effort to find the property. This last-chance set of properties is stored in a property of the constructor itself named `prototype`. If the constructor's `prototype` contains a property of the name we referenced, it will be returned as the value before the interpreter gives up the ghost and returns `undefined`.

This is an incredibly useful mechanism for sharing definitions among many instances of an object type without the need to copy the values into each and every instance.

At this point you might be thinking that this sounds a lot like class-level declarations of languages such as C++ and Java, and you'd be correct in thinking that there are high-level similarities. But the analogy can begin to break down pretty quickly if you start to dig into it, so it's probably best to think of the prototyping mechanism as a *defaulting* mechanism rather than as class-level declarations.

Defining our method as a `prototype` method of the CD object is simple, as shown in listing 3.6.

Listing 3.6 Using prototyping to define a method

```
function CD(title,artist,location) {
  this.title = title;
  this.artist = artist;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    this.location = location;
  }

  CD.prototype.whereIsIt = function() {
    alert( 'The CD is on shelf ' + this.location);
  }
```

Now, whenever an instance of CD is created, it essentially "inherits" any properties defined in the `prototype` of the constructor. Note that "inherits" was quoted quite purposefully, as this is *not* a true inheritance mechanism as defined by object-oriented concepts; it's a defaulting mechanism.

That distinction aside, prototyping is also useful for more than just methods. Any value that you'd like to define a default value for can be specified in the `prototype`. Consider for example, the following:

```
CD.protoype.LOCATION_PREFIX = 'The CD is on shelf ';
```

This essentially creates a "class constant" that can be referenced in any method of a CD instance, including the `whereIsIt()` method:

```
CD.prototype.whereIsIt = function() {
  alert( this.LOCATION_PREFIX + this.location);
}
```

If a particular instance wanted to provide another prefix, it could define a property with the same name. Since the interpreter will always look into the object instance first, any such declaration overrides a declaration in the `prototype`. One note of caution: once you define a property in an object instance, it will forever override any `prototype` declaration even if you set the property to `null`.

The built-in JavaScript objects are also `prototype`-based. As such, you can modify or extend them to suit your needs.

Although any such endeavor should be approached with all due caution—after all, you would be modifying classes intrinsic to the language—it's usually fairly safe to *add* convenience methods to the built-in types when methods you think would be useful are missing.

One such example could be a "trim" method on the String object. Trimming a string of trailing and leading white space is something that we might frequently be called upon to do—perhaps as part of validating user input into controls, where trailing white space could either need to be ignored or prohibited. Seemingly inexplicably, even though this is a very useful method and one most other languages with String classes provide, the String class in JavaScript has no such trimming function.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

So let's add one, shall we?

As we have seen, adding a method to a class's `prototype` is as easy as assigning a function literal to that `prototype`. Adding a `trim()` method to the String class is, then, as simple as shown in listing 3.7.

Listing 3.7 Teaching the String class how to trim

```
String.prototype.trim = function() {
  var matches = this.match(/^[ \t\n\r]+/);
  var prefixLength = (matches == null) ? 0 : matches[0].length;
  matches = this.match(/[ \t\r\n]+$/);
  var suffixLength = (matches == null) ? 0 : matches[0].length;
  return this.slice(prefixLength, this.length - suffixLength);
}
```

This new method to the String class uses JavaScript's regular expression matching capabilities to count the number of white space characters at the beginning and end of the string, and then uses the String's own `slice()` method to extract and return a trimmed version of the String's value.

Note that in a similar fashion to other String methods, the value of the String itself is not modified; rather, the modified version of the value is returned as the method's results. When extending objects, be they built-in or otherwise, it's always a good idea to follow the API style of the object when adding new functionalities.

In OO parlance, these methods that we've created, either on our classes or built-in classes like String, are called *instance methods* since they are accessed through instances of the class and possess the class instance as their function contexts.

But there's another type of method we can define.

Creating class methods

We saw how we can create instance methods by adding functions as properties to the `prototype` of constructors, but JavaScript also gives us the ability to create what would be called *class methods* in other OO languages. This is accomplished by adding a method, not to the constructor's `prototype` but to the constructor itself.

For example, these two statements perform very different actions:

```
Object.prototype.sayHi = function() { alert('Hi!'); }

Object.sayHello = function() { alert('Hello!'); }
```

The first statement creates a method named `sayHi()` accessible through each and every instance of type Object (which happens to be every object ever

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

created). The second creates a class method named sayHello() directly on the Object constructor.

In order to call the sayHi() method, an instance of the object is required so that its method can be invoked:

```
var o = new Object();
o.sayHi();
```

In order to call the sayHello() class method, we just need to use the owning constructor as a prefix:

```
Object.sayHello();
```

"Look, Ma! No instance!"

This type of method makes most sense when there is no need for an instance to be available to the code of the method as the function context. Indeed, in other OO languages, class methods have no concept of this, and any attempts to use this in one will usually result in a compilation error. However, in JavaScript *every* function has a function context, even class methods.

Without an object instance to serve as the function context, can you guess what does serve in that capacity? If you guessed the window, you get points for trying, but no cigar. For such methods, the *constructor function itself* serves as the function context. If the idea of a function serving as the function context blows your mind, don't worry too much about it. First, remember that in JavaScript, functions are a type of object and can serve in capacities that other objects can. Second, in this situation the fact that the constructor is the context object is pretty much useless. So it's OK to just ignore the fact that JavaScript class methods have a this reference if it makes you feel more comfortable.

Now that you know how JavaScript's mechanisms can be used to create JavaScript classes and other object-oriented constructs, let's put that knowledge to work and write a class of our own.

3.1.4 Writing a JavaScript class: a button

Now that you're aware of the JavaScript facilities for creating object-oriented classes, let's dig in and write a simple one.

The class we'll write in this section will be used to add instrumentation to an instance of an HTML button element. While the <button> element is a handy component of the set of HTML controls, it lacks the ability to present visual feedback to the user as it changes state.

The class we'll write will give us the ability to modify the visual rendition of the button when certain state changes occur, such as

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- When the mouse cursor enters the button area, thus "arming" the button
- When the mouse cursor exits the button area, "disarming" the button
- When the mouse button is pressed down, "pressing" the button
- When the mouse button is released, "un-pressing" the button
- When the button is enabled or disabled

By changing the CSS style class (not to be confused with an OO class) assigned to the button as these state changes occur, we can present visual feedback to the user beyond that provided by the native implementation.

We'll cleverly call our class *Button*, and by convention, define it in a JavaScript file named for the class it contains: `Button.js`.

To begin with, let's define some characteristics that we want this class to possess:

- The class will instrument an existing HTML `<button>` instance identified by its `id`.
- The class will accept a number of options at construction, consisting of
  – Whether the button is initially enabled or disabled
  – Style class names to be applied at the various state changes
  – A function to be executed when the button is clicked
- Reasonable defaults will be provided for any option not specified.

With all that in mind, let's begin by writing our constructor for this class.

The button constructor

As you'll recall from our earlier discussion, the *constructor* of a JavaScript class is simply a normal, everyday function. The distinction comes when the function intended to serve as a constructor is used as the operand of the `new` operator. This invokes the function with a context consisting of a newly allocated and empty Object instance.

By convention, a constructor is named for the class it creates, in our case Button, and should set up the object into an initial and valid state.

To achieve this, our button constructor has more than a few bits of information that it needs to accept in order to completely set up the instance. At minimum, it needs the `id` of the button that the instance is to instrument. According to its stated goals, it also takes a variety of optional information, including the names of various style classes and a callback handler to be invoked when the button is clicked.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The simplest approach is to just define a constructor with a parameter list that specifies each possible parameter. After all, we can supply null for any optional parameter that we don't want to include, right?

The signature of such a constructor would take the form of

```
function Button(elementName, disabled, onClickCallback,
                enabledStyleClass, disabledStyleClass,
                armedStyleClass, pressedStyleClass)
```

That's one required parameter (the element name), which is placed first in the parameter list, followed by six optional parameters.

OK, that's not too onerous to deal with from the point of view of coding the constructor; any parameters that come in as null just need to have default values provided for them.

But what about from the point of view of the *caller*?

Because association of parameter values to parameter names happens in an ordinal fashion, any function signature with a large number of optional parameters presents a not-so-nice API to users of that function. Consider an invocation of this constructor where the caller wants the initial state and all the style class names except the "pressed" style to default:

```
new Button('myButton',null,doSomething,null,null,null,
           'pressedButton');
```

Just look at all those nasty nulls.

By defining our constructor signature in this manner, we've forced the users of our class to count nulls carefully to be sure that they match the corresponding parameters in our fairly long parameter lists. And what if we decide to extend our class to encompass more functionalities and, therefore, more optional parameters?

It quickly becomes clear that defining function signatures with large numbers of parameters—particularly if many of those parameters are optional—is an unfriendly and unscalable solution. To avoid such ugliness, we'll adopt a technique that is rapidly gaining favor and acceptance for providing sets of optional parameters: the *hash*, sometimes called an *anonymous object*.

This is simply a single object, passed as a parameter, whose properties serve as the optional parameters to the function. Because they are passed as a single option, the number of parameters to a function is limited to the list of required parameters, followed by a single optional parameter. And because properties are named, the requirement to order the optional parameters in any particular sequence, or to leave blanks for parameters that are omitted, simply disappears.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This technique is widely used in popular JavaScript libraries (some of which we'll be taking a closer look at later in this and the next chapter), and reduces the rather unwieldy and less-than-readable constructor invocation that we saw earlier to

```
new Button('myButton',
           {
             onClick: doSomething,
             pressedClassName: 'pressedButton'
           });
```

Not only does this eliminate all those nulls, but it also makes for much more readable code as each optional parameter is explicitly named. There is no need to remember the order of parameters, and no counting is involved to figure out which parameter is which.

This does place a slightly larger burden on the constructor code, but that burden is minor and if there's going to be a burden anywhere, it's better to factor it into the constructor, which needs to be written *once*, than in each and every line of code that will ever be written to call that constructor.

OK, enough talk; let's code! Here's the skeleton for our constructor using the hash technique that we just described:

```
function Button(elementName, options) {
  //TODO: fill this in!
}
```

Now we'll fill in the body of the constructor. The workings of this code will be described in detail after listing 3.8.

Listing 3.8 Constructor for the Button class

```
function Button(elementName, options) {
  this.element = document.getElementById(elementName);       ❶ Locates HTML button
  if (!this.element) throw new Error(elementName + ' not found');
  this.element.button = this;
  this.options = options || {};                              ❷ Specifies options hash
  if (options) {
    this.options.enabled = options.enabled || true;
    this.options.onClick = options.onClick || function() {};
    this.options.enabledClassName =
      options.enabledClassName || this.CLASS_DEFAULT_CLASS_ENABLED;
    this.options.disabledClassName =
      options.disabledClassName || this.CLASS_DEFAULT_CLASS_DISABLED;
    this.options.armedClassName =
      options.armedClassName || this.CLASS_DEFAULT_CLASS_ARMED;
    this.options.pressedClassName =
      options.pressedClassName || this.CLASS_DEFAULT_CLASS_PRESSED;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  }
  var instance = this;                              ◁┐ 3 Instruments HTML button
  this.element.onclick = function() {
    if (instance.options.enabled) {
      instance.options.onClick.call(instance);
    }
  }                                                   4 Changes button's visual state
  this.element.onmouseover = this.onArm;           ◁┘
  this.element.onmouseout = this.onDisarm;
  this.element.onmousedown = this.onPress;
  this.element.onmouseup = this.onRelease;
  if (this.options.enabled) {                       ◁┐ Places button in
    this.enable();                                    5 proper state
  }
  else {
    this.disable();
  }
}
```

While this constructor may look a bit long and foreboding, what it does is actually fairly simple—but with some twists that might be confusing had we not examined them earlier in the chapter.

The very first thing we do in this constructor is to locate the HTML `<button>` element that we are going to instrument ❶. Using the `document.getElementById()` function, we obtain a reference to the element in the HTML DOM and assign it to a property on the context object named `element`. Remember that the `new` operator created a brand-spanking-new Object instance on our behalf and set it as the context object for our constructor. We don't have to worry about allocating the object; once the constructor is invoked, the new object is readily accessible via the `this` reference.

By assigning the DOM reference of the element to a property, we will be able to easily locate the `<button>` element whenever we have a reference to the Button object. If for any reason we can't locate the element whose `id` was passed, we throw an error that will appear on the JavaScript console of the browser being used. By doing so, an error on the caller's part (misspelling the `id` of the element, for example) results in a clear error rather than some mysterious "object has no properties" (or similarly nondescript) errors later down the line.

We now have a reference to the instrumented `<button>` element given a Button object instance, but we also want to be able to find the Button object instance given the `<button>` element. To facilitate that, we add a property to the `<button>` element named `button` that contains a reference to the Button instance.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This little trick means we can find either the DOM element or the object instance regardless of which one we happen to have a reference to. This will come in quite handy a little later on.

Having dealt with the `elementName` parameter, let's turn our attention to the options hash ❷. We are going to want to be able to access the options in the hash object throughout the methods of this class, so we'll assign the options object to a property of the object.

But wait! This hash is called "options" because, well, it's optional! What if the caller doesn't supply a hash object at all? We could be rather surly and insist that the caller supply an empty hash object when no options are to be passed, but that'd be downright unfriendly. Rather, we'll deal with the possibility within the constructor.

We do so with a statement that might look odd if you haven't seen this sort of construct before:

```
this.options = options || {};
```

Or-ing objects? Isn't that just for Boolean expressions?

This terse, but handy, notation essentially means "use this first operand if it exists, otherwise use the second." It works because, unlike in languages such as Java, any expression, Boolean or otherwise, can be used in a context where a Boolean is expected; coercion rules to express the values as Boolean are applied to each operand.

In our case, we use the fact that `null` and `undefined` convert to false. Therefore, if the first operand evaluates to either `null` or `undefined`, the expression evaluates to the second operand of the or-ed expression. So what the previous line does, in very succinct notation, is assign the `options` parameter to the property named `options` *if it exists*; otherwise, it assigns an empty object to the property.

At this point, `options` refers to an empty object if no properties were provided, or an object prepopulated by the caller with whatever options he or she specified. We now need to go through and assign reasonable default values to any option that was not specified. That *was* one of our goals, remember?

So, one by one, we test each possible option using the handy "or-ing" trick that we learned earlier. For each, we check if a value already exists, and if so, we leave it be. Otherwise, we assign a default value.

The first option we test is simple:

```
this.options.enabled = options.enabled || true;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If no `enabled` option was specified in the options hash, a default value of `true` is assigned.

Similarly, the next statement supplies a default callback function of the `onClick` handler:

```
this.options.onClick = options.onClick || function() {};
```

Here, since we have no idea what a reasonable action would be, we default to a function that does nothing if the caller supplied no callback.

The remainder of the options assign default CSS style class names in the absence of explicitly provided names. We'd probably expect the default to be a string containing the name, and it very well could be, but our implementation chooses a slightly different course.

The first of these statements is

```
this.options.enabledClassName =
  options.enabledClassName || this.CLASS_DEFAULT_CLASS_ENABLED;
```

The structure of this statement is the same as the others, but instead of a literal as a default value, we see a reference to something we know nothing about yet. Because it's referenced by `this`, we know it's going to be part of the definition of the class. But what?

These references will be properties that we will set up on the `prototype` for this class containing string literals that will define the default style class names. We don't just hard-code the string literals in the constructor because we're following a well-established practice. Factoring out such literals into "class members" serves two purposes:

- It isolates the string literals from the code, making them easier to locate for the programmer as opposed to embedding them in code where they may be hard to spot.
- If the methods in the class need to reference the string value more than once, the references all resolve to a single instance of the string. That way, simple typos in string literals don't turn into hard-to-find bugs, and any changes to the value of the literal in the course of development need to take place in *one* location.

The use of all uppercase characters when naming these references is a C++ and Java convention that is applied to "class constants," following the practice stated earlier. In JavaScript, there is no such thing as a *constant*, but by using this notation we make clear our intention that these values are to be treated as constants.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When these statements complete, we have a property that contains the combination of any caller-supplied options and class-supplied defaults. From this point forward, we have the luxury of having to make no distinction between the two; by constructing this options hash up-front, we have placed the entire configuration for our class in one neat little bundle for us to use throughout the remainder of the code.

So, next we turn our attention to actually instrumenting the HTML `<button>` ❸, starting with the `onclick` handler:

```
var instance = this;
this.element.onclick = function() {
  if (instance.options.enabled) {
    instance.options.onClick.call(instance);
  }
};
```

In this snippet, we assign an inline function as the `onclick` handler of the `<button>` element.

In this function, we need to check whether the button is enabled before firing off the click handler. (Actually, we could be lazy since this function should never be called for a disabled button, but you know what they say about an ounce of prevention!)

This poses a small problem for us. When an event handler is activated, its context object is the element that initiated the event; in this case, that would be the `<button>` element, and not our Button instance. We could either rely on the self-reference we placed on the element (something we will do in later methods) or employ the closure in which the function is defined.

Since we'll employ our self-reference in other methods, let's use the closure for this one if for no other reason than to demonstrate the use of closures. (In an actual implementation you'd probably want to pick one tactic and stick with it consistently, but for illustrative purposes, let's be daring.)

If you recall how closures work, the function that we assign to the `<button>` element's `onclick` handler will have access to variables defined in the scope in which the function is declared. However, the `this` reference is never included as part of a closure, and that's exactly what we need access to! So, we get around that little issue by assigning the value of `this` to a local variable named `instance`. As a local variable in scope when the function is declared, the value of `instance`, a copy of the reference to our Button instance, most certainly *is* available as part of the closure.

With click handling out of the way, let's turn our attention to the mouse events that we will want to capture in order to change the visual state of the `<button>`

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

element ❹. Unlike `onclick`, in which we assigned an anonymous inline function (and its closure) as the element handler, we assign references to methods that we'll define later in the `prototype`. Doing so makes for a cleaner-looking constructor and allows us to segment the code into smaller chunks, but we lose the ability to reference the instance of the Button object via closures:

```
this.element.onmouseover = this.onArm;
this.element.onmouseout = this.onDisarm;
this.element.onmousedown = this.onPress;
this.element.onmouseup = this.onRelease;
```

As we'll see in just a moment when we examine the implementation of these functions, the self-reference that we stored on the `<button>` element will come to our aid.

Finally, we want to make sure that the `<button>` element and Button instance are placed into the appropriate initial enabled or disabled state ❺. We call one of two methods depending on the state defined by the options hash. We'll investigate the implementation of those functions later, in listing 3.11.

Now, remember those "class constants" that weren't really constants at all? Let's define those next.

Class-level member variables

In our discussion of the constructor, we noted that it's often a good practice to factor literals (strings or otherwise) out of inline code and into members so that one instance of a literal can be consistently referenced throughout the code. In our Button class, we declare a series of string members to be used as the default values for the various CSS class names that will be assigned to the `<button>` element as it changes state. These are shown in listing 3.9.

Listing 3.9 Class-level members for CSS style class names

```
Button.prototype.CLASS_DEFAULT_CLASS_ENABLED = 'buttonEnabled';
Button.prototype.CLASS_DEFAULT_CLASS_ARMED = 'buttonArmed';
Button.prototype.CLASS_DEFAULT_CLASS_DISABLED = 'buttonDisabled';
Button.prototype.CLASS_DEFAULT_CLASS_PRESSED = 'buttonPressed';
```

We specify that we'd like these members to be treated as constants (even though JavaScript possesses no such concept) by using all uppercase names—a convention used in many languages such as Java, C, and C++ to indicate a constant reference. By collecting these string literals in this manner, we ensure that multiple references to the members reference the same string literal.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This is superior to hard-coding multiple instances of the literals throughout the code. A typo in the member reference will result in a relatively easy-to-debug "undefined" error, whereas a typo in a string literal results in the element not behaving correctly, thus leaving us to guess why until we spot the misspelled literal. Class-level members such as this can be used for either read-write variables that we want to share across multiple instances of Button, or for read-only pseudo-constants like the ones we have defined here.

With that behind us, we face defining the handler functions that we assigned to the mouse events of the `<button>` element. Let's take those on.

The mouse event handlers

Changing the visual state of the `<button>` element as the mouse events occur is a simple matter of swapping out different CSS style classes as the events occur. We've given the user of our class the option of supplying the style class names or using the default names that we provide.

As you'll recall, we assigned the mouse event handlers as references to methods of the Button class. For example, when the mouse cursor moves into the button area, we want to display the style class associated with the armed state.

We assigned the handler as such in the constructor:

```
this.element.onmouseover = this.onArm;
```

Now, we define that method (and its kin) as shown in listing 3.10.

Listing 3.10 Handling the mouse events

```
Button.prototype.onArm = function() {
  if (this.button.options.enabled) {
    this.className = this.button.options.armedClassName;
  }
}

Button.prototype.onDisarm = function() {
  if (this.button.options.enabled) {
    this.className = this.button.options.enabledClassName;
  }
}

Button.prototype.onPress = function() {
  if (this.button.options.enabled) {
    this.className = this.button.options.pressedClassName;
  }
}

Button.prototype.onRelease = function() {
  if (this.button.options.enabled) {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      this.className = this.button.options.enabledClassName;
    }
  }
```

Note that when any of these event handlers is triggered, its context object, despite being a method of Button, is the triggering element: in this case, the `<button>`. Yet we need access to the Button instance so we can look up the various options.

We could have dealt with the issue by defining the methods inline and using closures (as we did for the `onclick` handler), but since we thoughtfully placed a reference to the Button instance into a property on the `<button>` element during construction (creatively named `button`), we have the reference we need. So in each handler, we first check the `enabled` option to make sure the button is "live," and if so, we apply the appropriate style class to the element's `className` property.

The final step in defining our class is giving the caller control over the enabled state of the button.

The enabled state methods

We gave the user of our class the ability to initially place the button in either an enabled or a disabled state. A disabled button isn't much good to anyone if you can't enable it at some point, so we need to add methods to our class to let the caller have some level of control over this state postconstruction.

We'll define three methods for this purpose:

- Allow the caller to discover the current state.
- Allow the caller to place the button in an enabled state.
- Allow the caller to place the button in a disabled state.

The implementation of these methods is shown in listing 3.11.

Listing 3.11 Implementation of the enabled state methods

```
Button.prototype.isEnabled = function() {
  return this.options.enabled;
}

Button.prototype.enable = function() {     ❶ Places button in enabled state
  this.options.enabled = true;
  this.element.disabled = false;
  this.element.className = this.options.enabledClassName;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
Button.prototype.disable = function() {      <-2 Disables the button
  this.options.enabled = false;
  this.element.disabled = true;
  this.element.className = this.options.disabledClassName;
}
```

These methods are a bit more straightforward than their brethren. Because they are simple methods on the class (as opposed to event handlers), we don't need to worry about closures or event context objects; the context object of these methods is simply the Button instance, exactly as we'd expect.

With the `isEnabled()` method, we give the user of our class the means to determine the current state of the button by returning the value of the `enabled` option property. We will be careful to ensure that, in our remaining methods, this property always accurately represents the current state of the element.

In fact, let's take a look at the method for placing a button into the enabled state ❶. The first thing it does is to set the enabled options property to `true`. It then enables the `<button>` element itself by setting its disabled property to `false` (all of the HTML form elements use this reverse-logic setting to determine the enabled state of the controls). Finally, the CSS style class of the element is set to the enabled style class name.

The method to disable the button ❷ follows an identical set of steps, except it uses values that disable the button and place the appropriate style class upon it.

With that, our Button class is complete. Or at least until we think of something else to add to it!

In creating this small class, we employed many of the OO techniques that we discussed in earlier sections: object properties, functions, context objects, closures, and the `prototype` property. Armed with this knowledge, we are ready to embark on defining any other object classes that we'd like to create to help organize our code.

But before we get too far ahead of ourselves, we need to test our code to make sure that it works!

Testing the Button class

To test our Button class, we'll create a simple little HTML page and put the class through some of its paces. If all is correctly functioning, upon clicking an instrumented button on the test page we'd expect to see the alert shown in figure 3.4 that displays the ID of the clicked button element. Listing 3.12 shows us how to get there.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

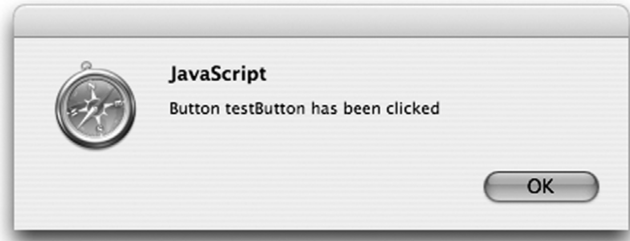

Figure 3.4
Did we get the right button?

Listing 3.12 Putting the Button class to the test

```
<html>
  <head>
    <title>Button Test</title>
    <script type="text/javascript" src="Button.js"> </script>      ❶ References .js file
    <script type="text/javascript">
      window.onload = function() {           ❷ Creates button instance
        window.testButton = new Button(
          'testButton',
          {
           onClick: onClicked
          });
      };

      function onClicked() {                 ❸ Triggers button
        alert('Button ' + this.element.id + ' has been clicked');
      }

      function toggleButtonState() {         ❹ Detects button state
        if (window.testButton.isEnabled()) {
          window.testButton.disable();
        }
        else {
          window.testButton.enable();
        }
      }
    </script>
    <style type="text/css">                  ❺ Sets visual styles
      #testButton {
        padding: 3px 6px;
        font-size: 1.1em;
        border-width: 3px;
      }
      .buttonEnabled {
        background-color: maroon;
        border-color: maroon;
        color: white;
        border-style: outset;
      }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      .buttonDisabled {
        background-color: #999999;
        border-color: #999999;
        color: white;
        border-style: outset;
      }
      .buttonArmed {
        background-color: maroon;
        border-color: maroon;
        color: orange;
        border-style: outset;
      }
      .buttonPressed {
        background-color: #660000;
        border-color: maroon;
        color: orange;
        border-style: inset;
      }
    </style>
  </head>

  <body>                                          Identifies <button> element ❻

    <button type="button" id="testButton">Click me!</button>   <-

    <div style="margin-top:16px">
      <input type="checkbox" onclick="toggleButtonState();">   <-
      Disable button
    </div>                                  Triggers enabling/
                                            disabling function ❼
  </body>

</html>
```

Note that this page in no way constitutes a thorough test of the class. It only serves as a template for a series of more thorough tests that you should consider conducting as an exercise. It does, however, represent the manner in which we believe that our Button class will be most often used.

In this page, we import our Button class by referencing the `.js` file ❶. This brings the Button constructor and its prototype into scope on the page. Note that we must do this for each page on which the Button class will be used.

In the `onload` handler for the page ❷, we create an instance of Button and assign it to a property named `testButton` on window. We do this so that the Button instance will be available throughout the rest of the page. Were we to simply use a `var`, its scope would be limited to the `onload` handler. We could also have declared a window-scoped `var` outside of the handler and assigned its value inside

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the handler, but why spread a declaration over two locations when you can make it one statement?

In the constructor for Button, we specified the `id` string of "testButton" that identifies a simple `<button>` element that is defined ❻ in the body of the page.

The only option we provide in this test is a reference to a function named `onClicked` that is to be triggered when the button is clicked.

That was easy, which is exactly what we intended. If the user needs to be a bit pickier, there are other options that can be specified. But this example probably represents how most Button instances will be created, and you can see that we've made things pretty easy for users of our class.

The next major element of our test page is the `onClicked()` function that will be triggered when a user clicks the `<button>` element ❸. In this function—which in a real environment would probably do something much more interesting—we issue an alert that verifies that the proper object was set as the context for this function.

The reference `this.element.id` verifies that the Button instance is the context object. The `this` reference should evaluate to the Button instance whose `element` property contains a reference to the `<button>` element.

That will test the button-clicking functionality, but we also want to exercise the enabled state functions of our class. To this end, we write a function ❹ that detects which state the button is in—using the handy method that we so thoughtfully provided—and set the button to the opposite state. We'll see in just a little while how we trigger this function from a page element.

The lion's share of the page is devoted to the CSS styles ❺ that give our button the visual feedback that we seek—after all, that was the whole purpose of the Button class. The styles chosen give the button an outset appearance when inactive and enabled, a "grayed-out" appearance when disabled, a highlighted appearance (by turning the text orange) when armed, and an inset appearance when pressed. These states are shown in figure 3.5, though they're not terribly exciting when shown in static screen shots (especially after conversion to grayscale), but you get the idea.

Figure 3.5 Button visual states

In the body of the HTML page, we define two control elements. First is the button ❻ that we are instrumenting. Note that the only attribute that we needed to

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

place on the button was its `id`. The remainder of the instrumentation, consisting of the handlers and class names, are applied by the Button class. How easy is that?

The second control element is a checkbox ❼ used to trigger the enabling/disabling function that we defined earlier ❹.

Obviously, the next series of tests that we should conduct would check that we can correctly override the default CSS class names with those of our own choosing, and verify that we can place the button into an initially disabled state.

Negative testing should also be conducted to ensure that our class behaves reasonably well when passed bad values. As an additional exercise, you might want to ponder how to make the class even more robust in this regard. For example, in the interests of brevity we never checked that the element passed was really a button! What other "holes" can you fill?

Now that we have a handle on how JavaScript enables us to write better-organized code using object-oriented principles, let's take a look at a popular JavaScript library that makes it easier for us to create such code.

3.2 The Prototype library

You've seen in the first part of this chapter how JavaScript is a language with built-in facilities for extending itself. The `prototype`-based inheritance feature of JavaScript makes it possible to create and extend not only our own classes, but also the classes that make up the language itself.

This ability has not escaped the attention of library writers, who have exploited these features to provide useful extensions to JavaScript. In this section we explore one such library, which has become very popular in the web development community: Prototype (so named after the `prototype` mechanism of JavaScript).

Prototype provides a wide array of extremely useful extensions and additions to JavaScript that make the life of web application authors, especially those writing applications containing a good amount of DHTML (such as Ajax applications), a lot easier. It has even become the basis for other popular and higher-level libraries available to JavaScript authors.

In this section we won't have the time or space to cover all of the features that Prototype has to offer; indeed, that could be the subject of a whole separate book. Rather, we'll concentrate on some of the most useful features, and then focus on those that will help us in our endeavor to organize our code using an object-oriented approach. In the next chapter, we will visit with Prototype again, this time with a specific focus on the Ajax features of Prototype.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

To use Prototype, all you need is the `prototype.js` file, which can be downloaded from the Prototype site at http://prototype.conio.net/. Import this library into your pages, and you're ready to go!

Let's begin by taking a look at some of the convenient features of Prototype that are useful to just about any JavaScript code.

3.2.1 Generally useful functions and extensions

Although our main goal here is to focus on the object creation and manipulation mechanisms of Prototype, there are a few general features of Prototype that are so useful—not to mention just plain nifty—that they deserve to be explored first.

We'll start by looking at a helper function that's so convenient, once you get accustomed to using it, you'll wonder how you ever got along without it.

Obtaining DOM element references

Whether our goal is to peek at its properties, add to or manipulate the property values in some fashion, or to apply full-on instrumentation as we did in the first part of this chapter, obtaining references to DOM elements in a page is a frequent occurrence.

This process usually entails locating DOM elements given their `id` value, so we'll fairly often write code along the lines of

```
document.getElementbyId("someElementId")
```

That's not difficult to grasp or to write, but it *is* rather wordy. Prototype defines a shorthand function for this very common operation in the guise of the `$()` function. (Yes, the name of the function is just the dollar sign.)

Using this function, element references by `id` become

```
$("someElementId")
```

This is a much terser notation than its native version. Granted, when first writing code using this function, it can take some getting used to. But once you've used it even a little bit, you get accustomed to the initially odd notation and find that you miss it badly in environments where you aren't using Prototype.

But more so than just offering a compact notation, the `$()` function has a well-planned nuance: if the value passed to the function is already a DOM reference (as opposed to an `id` string), the reference is simply returned as the result. This allows us to use `$()` without caring whether we have a DOM reference or an `id` string, and, perhaps more importantly, keeps us from having to care which we have. This may not sound like a big deal, but it will come in tremendously useful when writing classes as well as in general JavaScript code.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Element references aren't the only thing that Prototype makes it easier to obtain.

Getting the value of form controls

Another useful shorthand function provided by Prototype is the `$F()` function, which returns the *value* of the form control element whose `id` is passed as a parameter.

Let's say, for example, that we have a text control whose `id` is `someTextControl`. Its value could be obtained via

```
$F("someTextControl")
```

It is important to note that this function refers to control elements by their *`id`*, and not by their *`name`*. Form control elements possess both an `id` attribute and a `name` attribute, which can be rather confusing until you understand the difference between them. Consider the following control element:

```
<input type="text" id="someId" name="someName" value="whatever"/>
```

This element can be referenced in JavaScript code in two ways—by `name` (assuming that the element is contained within a form named `someForm`):

```
document.someForm.someName
```

or by `id`:

```
document.getElementById("someId")
```

The difference between the two attributes is that the `id` attribute assigns identification for the element in the HTML DOM and must be unique within the page, while the `name` attribute assigns the name of the request parameter that will be part of the request created when the form is submitted. Any number of controls can share the same `name`.

The `id` attribute is applicable to all elements that become part of the HTML DOM, while the `name` attribute is only applicable to control elements that represent values that will become parameters upon a form submission.

It's important to remember that the `$F()` function expects to be passed the `id` of the element, not the `name` of the element.

As a bonus, in typical Prototype fashion, the `$F()` function can be passed a reference to the DOM element for a control rather than its `id` and still function correctly.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3.2.2 Array extensions

It's hard to imagine programming without arrays. While there are other constructs that are useful for ordered lists of data, the array is one of the most prevalent and is intrinsic to most programming languages.

JavaScript is no exception. The Array class provides many useful methods for dealing with repeating elements of data in our pages. But, as they say, there's always room for improvement.

Prototype extends the JavaScript Array class with some very clever features that we will find useful in our applications. Again, we're only seeing the tip of the iceberg here, so please feel free to dig through the Prototype code or Internet resources for more details than we can cover here.

The $A() function

While not technically an extension of the Array object, the Prototype `$A()` function is useful for transforming other constructs into instances of the Array class. Of particular benefit to us as Ajax programmers is its ability to transform an XML document's `NodeList` into an Array. That way, not only can we easily traverse it, but we can also use it with the Prototype Array extensions that we're about to explore.

For example, let's assume we have an XML document referenced by a variable named `xmlDoc`:

```
var arrayOfNodes = $A(xmlDoc.getElementsByTagName('xyz'));
```

This would create an array of all `<xyz>` elements in the XML document and assign it to `arrayOfNodes`.

Now let's take a look at how Prototype has made the JavaScript Array class an even more useful construct than its native definition.

The Enumerable class and methods

Prototype extends the Array class itself with some interesting functions such as `shift()` (which allows you to treat an array as a stack) and `compact()` (which will return a copy of a source array with undefined entries removed). But what will really interest us as we progress through Ajax examples in this book is that Prototype makes Array an "extension" of a Prototype-defined class named *Enumerable*.

We use the term "extension" loosely here because, while we know that JavaScript has no true object-oriented extending capabilities, Prototype allows us to fake it rather admirably. (We'll be talking about that in section 3.2.5.)

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The Enumerable class features a bevy of useful functions for iterating over the elements of an Enumerable instance (and hence, all arrays), the simplest of which is the `each()` function.

This function accepts a single parameter, a reference to an iterator function that will be invoked for each element of the Enumerable instance in indexed order. This iterator function should adhere to the following interface:

```
function iteratorFunction(element,index);
```

where `element` is the current element from the array, and `index` is that element's index within the array. A trivial example is shown in listing 3.13.

Listing 3.13 Using the `Enumerable.each()` method on an Array

```
var myArray = [ 1, 2, 3, 4 ];
myArray.each(showMe);              <- Iterates over each myArray element

function showMe(element,index) {   <- Is invoked for each element
  document.write('<p>[' + index + '] ' + element + '</p>');
}
```

Although the usefulness of this function may not be evident in this simple example—after all, would it not be just as easy, and perhaps clearer, to just use a `for` loop?—its utility will become clear when employed for more complex tasks. Never underestimate the benefit of factoring complex code out of an algorithm into a separate function—especially if the function is well named. The simplicity that such factoring can bring to complex code is usually well worth the time spent in structuring it properly.

We'll see in later sections how using this pattern helps to keep the individual functions shorter and easier to mentally grasp. Using this pattern also reduces the amount of notation needed to address the array elements when processing them.

The Enumerable class sports many more such iteration functions that can be used to make our way through arrays in a customizable fashion, and even to easily create new arrays by applying various criteria to the elements to choose which should become part of the new array. You are strongly encouraged to explore these functions.

Arrays, as we stated, are an essential element in programming for storing ordered lists of data. Let's take a look at how Prototype gives us a formalized version of another essential construct that we've already examined within this chapter: the hash as implemented by the Hash class.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

3.2.3 The Hash class

As we discovered earlier in this chapter, the fact that object properties can be dynamically created and assigned values allows us to use instances of the Object class as ad hoc "associative arrays," which are similar to what we think of as Maps or Hashes in other languages.

Prototype takes this one step further by formally defining a *Hash* class that not only provides the expected key-to-value associations, but also features such useful methods as `keys()`, `values()`, and `merge()`. It also defines a method that we will find incredibly useful in Ajax code: the `toQueryString()` method.

This method formats and returns an HTTP-compliant *query string* formed from the name-value pairs contained within the hash. Together with the `$H()` function, which creates an instance of Hash from the properties and values of any JavaScript object, the `toQueryString()` method will be employed in later examples to help us easily generate, with minimum chance of error, URLs to use as Ajax targets.

Listing 3.14 shows a small example of its use, which would result in the following output:

```
a=1&b=2&c=3
```

Now *that's* nifty! Even better, this function handles any URL encoding that might be necessary for reserved characters in the names or values of the query string parameters.

Listing 3.14 Using the `Hash.toQueryString()` method

```
var o = new Object();
o.a = 1;
o.b = 2;
o.c = 3;
document.write($H(o).toQueryString());
```

Creates object, assigns properties

Formats and writes query string

Consider the previous example with more complex values and alternative notation. It displays

```
param%201=1%261&param%202=2%3D2&param%203=3%2B3
```

Here's the code that makes it happen:

```
var queryString = $H(
  {
    'param 1': '1&1',
    'param 2': '2=2',
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      'param 3': '3+3'
    }
  ).toQueryString();
  document.write(queryString);
```

Note that in this example, each parameter name contains a space character that needs to be encoded, and each value likewise contains characters (&, =, and +) that also need encoding.

Each special character has been automatically replaced with its encoding. Aren't you glad you won't have to do all that yourself?

3.2.4 Binding context objects to functions

Recalling our discussion of functions in section 3.1.2, remember that the context object of a function is the object that is referenced by the `this` reference while that function is executing. Normally, this object is the page's window (for top-level functions), or the owning object instance when functions are invoked as methods.

But when methods of an object are invoked as callback (handler) functions as the result of an event (such as a mouse click), the `this` variable refers to the event-generating element. Under these circumstances we employed some JavaScript sleight-of-hand to make sure that we had access to the instance of the method's object.

One means that we can use is to invoke the function using the Function class's `call()` method, rather than making a *direct* call to a function. When we do so, the first parameter specified to `call()` is established as the context object for the function.

For example, if we want the `this` variable within a called method to refer to an object of our choosing, we would employ the function's `call()` method to set the object that will become `this` within the function:

```
var a = someFunction.call(someObject);
```

By doing so, the `this` reference in the body of the `someFunction()` function will refer to `someObject`.

That's all well and good when *we* are the ones calling the function. But what about when we do not have control at the time of the call, for example, when passing a function reference to another object to use as a callback? That's a very real scenario that we'll encounter within our Ajax programs in which we pass function references to XMLHttpRequest or other library code to be used as callback notifications. We'll no longer have control over how the function will be invoked, and yet we'd like to bind the `this` variable to a specific object other than the one to which it would normally be bound.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For just these occasions, Prototype has extended the Function class with a method named `bind()`, which we can use to pre-bind an object to the function reference such that, when the function is later invoked, its `this` variable will point to that object.

Once again, a listing being worth a thousand words, see listing 3.15 for an example.

Listing 3.15 Pre-binding an Object instance to a function

```
window.x = 1;                       <-- (1) Marks the window object
var o = { x: 2 };                   <-- (2) Creates and marks an arbitrary object

function doSomething(callback) {    <-- (3) Defines unbound callback
  callback();
}

function callback() {               <-- (4) Defines bound callback
  alert(this.x);
}

doSomething(callback);
doSomething(callback.bind(o));
```

In this example, we mark the window object with a property named `x` (1) and create an arbitrary object, also assigning it a property `x` but with a different value (2). This will allow us to easily identify to which of these objects a reference is pointing.

We then define a processing function named `doSomething()` that accepts a callback function referenced as its lone parameter (3). For illustrative purposes, this function does nothing but invoke the passed callback function, but in a real-world example, it could be a library function that performs some processing and then invokes the callback when it is through.

The callback function itself (4) merely displays an alert with the value of the `x` property for whatever object is referred to by the `this` variable.

To demonstrate the difference between an unbound and bound function, we first call the `doSomething()` function, passing in a simple reference to the callback function. When the alert is displayed, we see the value `1`, indicating that for the invocation of the callback, the `this` variable points to the window object.

Then, we call `doSomething()` again, this time binding the object `o` to the callback using Prototype's `bind()` extension. When the alert appears, we see the value `2`, indicating that the `this` pointer now refers to the `o` object.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This type of pre-binding will turn out to be incredibly useful to us when passing handler functions to the processing methods of XHR and other library functions, thus allowing us to control what the *context object* (the object referred to by `this`) will be when the handler is invoked.

When your callback function is going to be invoked as the result of a DOM event (such as a mouse click) and that function needs access to the triggering event, Prototype provides a similar method named `bindAsEventListener()`. This method operates in a similar fashion to `bind()`, accepting as its parameter the object that is to be established as the context. However, it makes sure that when the callback is invoked, the event object is passed as the parameter to the callback. We'll see examples of this method's use in later chapters.

With all this newfound Prototype know-how, now let's take a look at how Prototype helps us to write object-oriented JavaScript.

3.2.5 Object-oriented Prototype

We've seen in the first part of this chapter how applying object-oriented concepts to JavaScript can help make code better-organized and reusable. In this section we will explore the way that Prototype makes it even easier for us to write object-oriented JavaScript classes.

Let's start off by looking at a Prototype-provided class named Class.

Creating classes with Prototype

Prototype's Class class is a shorthand means for generating object constructors using a simpler notation—one that is also perhaps more consistent—than when using raw JavaScript.

As you may remember from our Button example of section 3.1.4, a JavaScript class definition consists of a constructor function followed by declarations that assign members and methods to the `prototype` of that constructor. Some could find that notation a bit inconsistent, with initialization code going in a normal function declaration that operates as the constructor while method code is placed in functions assigned to the `prototype`.

For those who would prefer to aggregate the code in a more consistent manner, the `create()` method of Class (which turns out to be its only method) generates a constructor whose functionality can then be declared in the `prototype`. When the constructor is invoked, it will hand control over to a method named `initialize()` that it will expect to find defined within the class. Let's take a look at the example in listing 3.16.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 3.16 Constructing a class the Prototype way

```
Something = Class.create();

Something.prototype.initialize = function(p1,p2,p3) {
  /* constructor code goes here */
};

Something.prototype.someMethod = function() {
  /* method code goes here */
};
```

In listing 3.16 we see that the code to construct an instance of the object is delegated to a method named `initialize()`, which can be passed any number of parameters and is declared just like any other method in the `prototype` of the class. Many developers prefer this consistency, in which all code is declared in the `prototype`.

For even tighter-looking code, many developers also use the alternate JSON notation for aggregating the `prototype` properties, as shown in listing 3.17.

Listing 3.17 Aggregating properties the JSON way

```
Something = Class.create();

Something.prototype = {

  initialize: function(p1,p2,p3) {
    /* constructor code goes here */
  },

  someMethod: function() {
    /* method code goes here */
  }

}
```

The code shown in listing 3.17 is entirely equivalent to the code in listing 3.16, albeit using a different notation. Many developers prefer to assign the members and methods to the `prototype` for a class en masse in this manner.

Which notation you use to declare your JavaScript classes is a matter of personal preference. Regardless of how you declare the class—using raw JavaScript, or with the assistance of Prototype's Class, and with or without using JSON notation—the remainder of Prototype's object extensions can be used. And regardless

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

of the means of declaration, instances of the class are created in the same manner using the `new` operator:

```
var anInstance = new Something(1,2,3);
```

Prototype has many extensions that make the various things that we need to do with objects easier to manage. Let's take a peek at a few of them.

Merging objects with Prototype

Whether you've realized it or not, we've already looked at an instance of merging objects earlier in this chapter. But before we get into that, let's take a look at how Prototype allows us to merge objects and exactly what that means.

In Prototype, the concept of merging two objects is to essentially make a union of all the properties found in both objects. This is accomplished with a class method defined on the Object constructor named `extend()`.

Odd. Wouldn't you expect it to be named `merge()`? Well, the reason for the choice of name will become clear before too much longer.

The `extend()` function is destructive in that one of the two parameters passed to it is modified to be the result of the merge. The signature of the method is

```
Object.extend(object1,object2)
```

The function operates by copying any properties found in `object2` into `object1`. The result is that `object1` ends up with all the properties it initially possessed, as well as all the properties that are in `object2`.

If both objects possess a property with the same name, the `object2` property value is copied *over* the `object1` property value, giving the `object2` properties precedence. When the merge is complete, a reference to `object1` is returned as the value of the function.

When we rewrite our Button class using Prototype's assistance, we'll find this method useful for dealing with the options hash. We'll see all that in section 3.2.6.

Beyond merging object instances, the `extend()` method has a more fundamental use—one that explains its name.

Extending classes with Prototype

In object-oriented languages such as Java and C++, class hierarchies can be created through *inheritance*, in which a subclass inherits members and methods from a superclass. JavaScript possesses no such inheritance capabilities, but the `Object.extend()` class method gives us a darn good approximation of those facilities.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Remember that a JavaScript class is created by the combination of a constructor and properties defined in its `prototype`. While we can't magically cause a JavaScript class to inherit anything from another class, what if we use the `Object.extend()` method to merge the `prototype` of an object serving as a subclass with a superclass object to form a new `protoype` composed from both objects?

Head spinning yet?

Let's take a look at an example to see if we can make this work.

Remember our CD example from the beginning of the chapter? We created a small object to hold information that represented a CD in our vast collection. We recorded (as properties) the title of the CD, the artist, and the location (as a shelf number) where the physical disc is stored.

Well, as it turns out, we're not only wild about music, we're also movie buffs! So we'd like to expand our example to also include DVDs. We're just crazy that way.

As we know, CDs and DVDs share a lot of characteristics, but they each possess unique characteristics as well. The concept of an "artist" as applied to a CD doesn't make much sense for DVDs, and with DVDs we might want to record the director of the film, which makes no sense for CDs. But both share the characteristics of a title and a location in our collection.

Let's start by creating a class that describes the common characteristics of both of these types of discs. Using Prototype, the result is shown in listing 3.18.

Listing 3.18 Defining the Disc "superclass"

```
Disc = Class.create();          <-- ❶ Creates the constructor, Prototype style

Disc.prototype = {

  initialize: function(title,location,type) {     <-- ❷ Defines initialize method
    this._initializeDisc(title,location,type);          for constructor
  },

  _initializeDisc: function(title,location,type) {
    this.title = title;
    this.location = location;
    this.type = type;
  },

  whereIsIt: function() {
    return 'The ' + this.type + ' titled ' + this.title +
           ' is on shelf ' + this.location;
  }

}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

As you saw earlier, first we create the constructor (the Prototype way) ❶, and then define the members in the `prototype`. Because we used Prototype's `Class.create()` mechanism, we define an `initialize()` method ❷ for the constructor to call in order to set up the instance at construction time. And that's where we did something just a little bit odd.

The `initialize()` method just turns around and calls yet another method named `_initializeDisc()`, and lets *it* set up the instance. What's up with that?

First of all, the leading underscore in the name of the `_initializeDisc()` method is just a convention used to indicate that the method is intended to be used internally and should never be called by code that employs this class. JavaScript doesn't possess the concept of private or protected members, so by naming the method in that way, we indicate our *intention* that the method be ignored by code outside of this class (or its hierarchy, as we will see) even if we have no way to actually *enforce* it.

But why further delegate in this manner at all?

Remember that our intention is that this class serve as a superclass for the yet-to-be written CD and DVD classes. When we set up those classes, each will have its own `initialize()` method that will supercede the one we are defining in this class. To make sure that we can initialize the superclass from the subclasses, we factor the initialization code into the `_initializeDisc()` method, which will *not* be superceded by the subclasses, thereby keeping it available for the subclasses to call. We need to do this sort of two-level initialization for any class that is to be used as a superclass to be extended by other classes.

You might consider `_initializeDisc()` to be rather wordy, or feel that the use of the class name in this local initializer is redundant, but consider a situation in which the subclasses are to be used as superclasses for yet *other* classes. If we used a simpler name such as `_initialize()`, we run into the same problems where that method would be superceded by classes further down in the inheritance chain. By using the class name as part of the name for the local initializer, each class has a unique name for its local initializer with no chance of it being superceded by a subclass.

OK, so now that we know how to properly code a class so that it can be used as a superclass, let's see how we code the subclasses: CD and DVD. Listing 3.19 shows the code for the CD subclass.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 3.19 Coding the CD subclass

```
CD = Class.create();                                    ❶ Defines Propotype's
                                                          Class.create() method
CD.prototype = Object.extend(                           ❷ Merges with Disc class
  new Disc(),
  {
    initialize: function(title,artist,location) {
      this._initializeDisc(title,location,'CD');       ❸ Calls "inherited"
      this.artist = artist;                               initializer
    }
  }
);
```

Listing 3.19 is surprisingly short, but upon examination, not so simple. There are a few strange things going on here.

The first thing we do is to use Prototype's `Class.create()` method in the normal fashion ❶. But when we get to defining the class `prototype`, things get considerably more interesting.

Normally when creating a class `prototype`, we assign an object with properties representing the members and methods of the class to the class's `prototype`. And we do that here, *except* that instead of assigning it directly, we use it as the second parameter to the `Object.extend()` method, merging it with a new instance of the Disc class ❷.

What's going on there?

If we recall how `Object.extend()` operates, it takes all the properties it finds on the object passed as the second parameter, adds them to the object passed as the first parameter, and returns that object as the result of the method.

So here's what's happening: the hash object that we use to define the members of CD (in this case, solely consisting of the `initialize()` method) is added to a new instance of Disc, and that Disc instance becomes the `prototype` object for CD. The result is that the CD `prototype` contains all the members of Disc, and all the members of CD. Because CD is the second parameter to `Object.extend()`, any properties that it has in common with Disc will be given precedence.

In reality, no inheritance has occurred. But by merging the properties of CD with those of Disc, the *perception* of inheritance is achieved: it appears that CD has not only defined its own properties, but has also inherited Disc's properties.

The other interesting aspect of this example to note is that in the initializer for CD, we call the local initializer that we inherited from the Disc class to set the common properties ❸.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The declaration of the DVD subclass is similar, as you can see in listing 3.20.

Listing 3.20 Coding the DVD subclass

```
DVD = Class.create();

DVD.prototype = Object.extend(
  new Disc(),
  {
    initialize: function(title,director,location) {
      this._initializeDisc(title,location,'DVD');
      this.director = director;
    }
  }
);
```

Shall we do some rudimentary testing? The result is shown in figure 3.6. Listing 3.21 shows a simple test page that created this output.

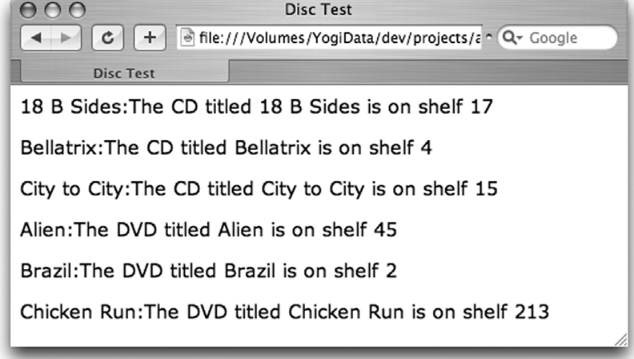

Figure 3.6 Where are the discs?

Listing 3.21 Testing the hierarchy

```
<html>
  <head>
    <title>Disc Test</title>
    <script type="text/javascript" src="../scripts/prototype.js"> </script>
    <script type="text/javascript" src="Disc.js"> </script>
    <script type="text/javascript" src="CD.js"> </script>
    <script type="text/javascript" src="DVD.js"> </script>
    <script type="text/javascript">
      var myCollection = [
       new CD('18 B Sides','Moby',17),
       new CD('Bellatrix','Jorgen Skogmo',4),
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        new CD('City to City','Jerry Rafferty',15),
        new DVD('Alien','Ridley Scott',45),
        new DVD('Brazil','Terry Gilliam',2),
        new DVD('Chicken Run','Nick Park',213)
        /* and on and on ... */
      ];
    </script>
  </head>

  <body>

    <script type="text/javascript">
      myCollection.each(
        function(disc) {
          document.write(
            '<p>' + disc.title + ':' + disc.whereIsIt() + '</p>'
          );
        }
      );
    </script>

  </body>

</html>
```

Naturally, if we were going to actually record our collection, we'd want to do so in a database rather than in an HTML page! But we're doing it here to illustrate the concept of using `Object.extend()` to emulate class inheritance, so pretend this is all reasonable.

When the page is displayed, we'd expect to see the name and locations of the discs in our collection, as shown in figure 3.6. Note the use of the `whereIsIt()` method, which was inherited by CD and DVD from Disc.

Before we move on to the next chapter—which takes a look at how Prototype and a handful of other freely available tools will help us to easily write Ajax code—let's use the Prototype facilities we've learned about so far to rewrite the Button class we developed earlier in the chapter.

3.2.6 Rewriting the Button class with Prototype

With the sleight-of-hand tricks that Prototype provides us for declaring JavaScript object classes, let's reimplement the Button class from section 3.1.4 using these newfound abilities.

The rewrite contains a lot of code that looks familiar, but it also has a number of significant changes, as you can see in listing 3.22.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 3.22 The Button class revisited

```
Button = Class.create();          <-- 1 Declares the constructor

Button.prototype = {              <-- 2 Declares class prototype
                                      3 Accepts constructor's
  initialize: function(element, options) {    <-- parameters
    this.element = $(element);
    if (!this.element) throw new Error(element + ' not found');
    this.options = Object.extend(
      {
        enabled: true,
        onClick: function() {},
        enabledClassName: this.CLASS_DEFAULT_CLASS_ENABLED,
        disabledClassName: this.CLASS_DEFAULT_CLASS_DISABLED,
        armedClassName: this.CLASS_DEFAULT_CLASS_ARMED,
        pressedClassName: this.CLASS_DEFAULT_CLASS_PRESSED
      },
      options
    );
    this.element.onclick = this.onclick.bind(this);
    this.element.onmouseover = this.onArm.bind(this);
    this.element.onmouseout = this.onDisarm.bind(this);
    this.element.onmousedown = this.onPress.bind(this);
    this.element.onmouseup = this.onRelease.bind(this);
    if (this.options.enabled) {
      this.enable();
    }
    else {
      this.disable();
    }
  },

  CLASS_DEFAULT_CLASS_ENABLED: 'buttonEnabled',
  CLASS_DEFAULT_CLASS_ARMED: 'buttonArmed',
  CLASS_DEFAULT_CLASS_DISABLED: 'buttonDisabled',
  CLASS_DEFAULT_CLASS_PRESSED: 'buttonPressed',

  onclick: function() {
    if (this.options.enabled) {
      this.options.onClick.call(this);
    }
  },

  onArm: function() {
    if (this.options.enabled) {
      this.element.className = this.options.armedClassName;
    }
  },

  onDisarm: function() {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      if (this.options.enabled) {
        this.element.className = this.options.enabledClassName;
      }
    },

    onPress: function() {
      if (this.options.enabled) {
        this.element.className = this.options.pressedClassName;
      }
    },

    onRelease: function() {
      if (this.options.enabled) {
        this.element.className = this.options.enabledClassName;
      }
    },

    isEnabled: function() {
      return this.options.enabled;
    },

    enable: function() {
      this.options.enabled = true;
      this.element.disabled = false;
      this.element.className = this.options.enabledClassName;
    },

    disable: function() {
      this.options.enabled = false;
      this.element.disabled = true;
      this.element.className = this.options.disabledClassName;
    }

  }
```

This rewrite features some significant differences from the original implementation. First, we used `Class.create()` to declare the constructor ❶. Again, Prototype's approach to declaring a constructor in this manner—moving the actual setup code into an initialization method—lends a certain consistency to the class code that you might find desirable. If not, you can continue to use the original native notation as a matter of taste. Regardless of which notation is used, the remainder of Prototype's facilities can still be used.

Next, the class's `prototype` is declared using JSON notation ❷. Use of this notation is orthogonal to the use of Prototype; it's just an alternative notation we can choose to use—or not to use. But this notation *does* appear to be the choice

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

of many developers who use Prototype—and that's where things start to get really interesting.

Because we used Prototype's constructor mechanism, we need to create an `initialize()` method that accepts the constructor's parameters. We do so ❸, and in the very first line we assign the `element` parameter to the `element` member. Because we use Prototype's `$()` function, the `element` parameter can be *either* an element `id` or a reference to the element.

We could have coded this sort of flexibility into our original class by doing some type checking and conditional assignment, but here we get it for free by using `$()`. We also renamed the parameter from the original `elementName` to the more general `element` to indicate this.

Unlike in our original implementation, we do not create a property on the `<button>` element that points back to this instance of Button. We're not being stubborn; it's just that it's going to turn out not to be necessary. We'll see why in just a bit.

When it comes time to set up the options, Prototype's assistance really starts to shine. In the original, we populated the `options` member one option at a time, testing to see if the caller had provided a setting for the option and using a default value if not. In our new implementation, we use the power of `Object.extend()` to merge the caller-provided options object with one of our own that we prepopulate with the defaults.

Compare the notation used in the original in listing 3.8 to that in listing 3.22. The clarity of the latter approach should be more than apparent. The set of options available and their default values is much better organized and clear at a moment's glance.

Next, we assign the `<button>` elements handlers. Using the `bind()` extension that Prototype added to the Function object, we quickly and easily set up each handler so that when called, their function contexts will be the Button instance rather than the `<button>` element.

In our native implementation, either we relied on closures to get a reference to the Button instance, or we relied on the property we had added to the `<button>` element. Neither of these tricks is needed here, which is why we could dispense with adding any property to the element.

The constructor is completed with the exact same code as the original to ensure that the button is in the correct initial state.

The remainder of the implementation is straightforward and simple. It is important to note that (thanks to the use of `bind()`) when the event handlers are

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

set up, all the handlers can assume that the Button instance (and not the `<button>` element) is the context object when the handler is triggered.

3.3 Summary

We presented a great deal of material in this chapter.

We focused on how JavaScript code can be organized using object-oriented concepts even though the language doesn't natively provide some object-oriented facilities common to other OO languages. You learned how to create JavaScript classes that contain members and methods, and by doing so gained all the benefits that JavaScript's object-oriented brethren lend to code written in those languages. Organizing code into classes not only makes it, well, more organized, but also facilitates reuse and will help the code to be more extensible and maintainable.

We then introduced the Prototype library. You saw that Prototype provides a bevy of helpful functions that could be useful in just about any JavaScript code base. We then looked into how Prototype helps us to write object-oriented JavaScript easily and more clearly. You saw how to merge objects, and how to use that ability to emulate class inheritance in a language that possesses no such concept.

But bear in mind that we've only scratched the surface here. There's much more to Prototype than we are able to cover in half a chapter. For example, Prototype contains facilities for easing the implementation of event-handling code—a usually onerous task due to the browser-specific nature of events. However, we're not quite done with Prototype yet. We will see a bit more of Prototype, in particular its Ajax facilities, in the next chapter.

While Prototype is fast becoming one of the most popular JavaScript libraries, it's far from the only one. In the next chapter we explore how some of the various freely available libraries (to include Prototype) can specifically help us as writers of Ajax applications.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

4 Open source Ajax toolkits

This chapter covers

- Choosing an open source toolkit
- Making Ajax requests using the Dojo toolkit
- Making Ajax requests using Prototype
- Making Ajax requests using jQuery
- Invoking server-side Java methods with DWR

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Sometimes other people make our lives easier. Sometimes we pay them for this service. And sometimes we can actually get something for nothing.

The Internet is full of open source tools that people have made available for others to use without charge, free for the taking. Whether the motivation behind making their labors freely available is a matter of seeking recognition, resume building, free advertising for other services, bragging rights, or just plain old-fashioned altruism, we can gratefully take advantage of these tools.

Though that doesn't mean we get an entirely free ride.

Since just about anybody with an FTP client can put just about anything out there on the Net, it behooves us to carefully choose which library or tool we are going to make use of. We need to take any number of factors into account when choosing open source software to use, but one good indicator is the number of successful projects that have already employed a tool or library.

In this chapter, we'll survey a few of the open source toolkits that can make our lives—as Ajax web developers—a bit easier. We'll look at the Dojo toolkit, Prototype (again, this time with an eye toward its Ajax capabilities), jQuery, and DWR. You will find all of these libraries used in examples throughout the remainder of the book. Prototype will be used extensively, while the other libraries are used here and again among the examples.

Be aware that the descriptions and examples within this chapter are in no way intended to serve as complete tutorials or primers for the full set of features of the toolkits that we'll examine. Nor will a comprehensive survey of the toolkits' capabilities be presented. Rather, we'll focus on the aspects of the toolkits that simplify the process of asynchronously communicating between client browser and server using Ajax.

Examining the remainder of the features each toolkit brings to the table will be an exercise in discovery—usually one that's really fun—left to the reader.

4.1 The Dojo toolkit

Dojo is an open source JavaScript library published and maintained by the Dojo Foundation. Like most other JavaScript toolkits, its aim is to make DHTML tasks—especially popular but complicated tasks such as animations—easier to create and maintain.

You can download an *edition* of Dojo at http://dojotoolkit.org/download/. Dojo is packaged as numerous editions that contain specific packages that you might be interested in using. For the purposes of the code that we'll write using Dojo, any edition that includes the I/O package will do.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

4.1.1 Asynchronous requests with Dojo

After downloading, simply place the `dojo.js` file in an appropriate location within your web application. For ease of setup, we'll just put it in the same folder as our examples for this section.

With regard to making asynchronous requests to the server, Dojo provides a simplification of the steps needed to make and respond to Ajax requests in the guise of the `dojo.io.bind()` function. This function accepts a single parameter: a JavaScript object whose properties serve as the parameters to the function—similar to the options hash that we examined in the previous chapter. This may seem to be a rather unconventional approach if you haven't come across it before, but it does have some distinct advantages and is gaining popularity among JavaScript programmers and especially toolkit authors. Using this technique, parameter order becomes moot, optional parameters are easy to deal with, and since each parameter property is named, the calling syntax is highly readable.

Problem

We want to create a page that we can use to look up phone numbers, given a list of names. We want to do so without any form submission that requires us to refresh the page.

Solution

For this problem, we're going to make use of the Dojo I/O bind function to make the asynchronous call to the server in order to look up and return a phone number, given a name string.

First, we need to import the Dojo library. In the head element of our page, we add

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

Then, we'll hard-code the list of names in order to keep this example simple. (In a later problem, we'll look at ways of obtaining such lists dynamically).

In the body section of our page we write the following:

```
<form name="lookupForm" onsubmit="return false;">    <-❶ Declares form element
  <select name="who" onchange="lookup();">
    <option value="JOHN">John</option>     <-❷ Hard-codes control selections
    <option value="MARY">Mary</option>
    <option value="BILL">Bill</option>
  </select>
</form>
```

We declare a form element ❶ with an `onsubmit` event handler that prevents the form from ever being submitted to the server. We're handling the server traffic

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

ourselves, so we don't ever need the form to be submitted—at least not in this example. A select element is declared with an `onchange` event handler that will trigger the lookup of the phone number when a selection from the list has been made.

The options in the control are hard-coded on the page ❷ for this example. Obviously in real-world code these would need to be dynamically created from the names available in the server's contact database. We can either gloss over that for this example, or imagine that this page was set up by some server-side mechanism, JSP or PHP perhaps, that handled that aspect of the page for us.

Upon selection, the `lookup()` function is invoked as the element's `onchange` event handler. This is where we use Dojo to make the Ajax call on our behalf:

```
function lookup() {
  dojo.io.bind(
    {
      url: 'phone.jsp?who='+document.lookupForm.who.value,
      mimetype: 'text/plain',
      load:
        function(type, data, req) {
          document.getElementById('displayArea').innerHTML = data;
        }
    }
  );
}
```

In this function, we make a call to the `dojo.io.bind()` function, passing all the information it needs in order to process our request as a JSON-formatted JavaScript object with specific properties that act as the function's parameters.

The `url` property specifies the URL of the server-side resource to invoke. In this example, we've defined a simple JSP file to handle the request. This ridiculously simple JSP file, which takes advantage of the powerful JSP Expression Language, merely takes the `who` request parameter that we pass to it and adds it to the end of a phone number prefix in order to create the phone number. It consists of the single line

```
555.555.${param.who}
```

It's easy to envision, however, that this JSP (or servlet, or PHP script, or other server-side resource) could just as well perform a database lookup or other process in order to obtain the phone number.

By default, the `dojo.io.bind()` function will use the GET HTTP method when making the request to the server. We could change this by including the `method` property with a value of `POST` and using the `content` property to provide the request parameter.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The `mimetype` property is set to specify that the response will be plain text. The `load` property specifies a handler function that is to be invoked upon normal completion of the request. Since the handler function is so short in this example, we've inlined it as part of the parameter object. It could just as readily be a reference to a function defined elsewhere on the page. If we wanted to register a handler to be called in the event of a problem, we could do that with the `error` property.

The `load` handler function is passed three parameters:

- The first is the type of handler being invoked. In this case it will always be `load`. This parameter allows a handler function to be reused for more than one event type.
- The second parameter is the response data—in this case, the generated phone number.
- The third parameter is a reference to the XMLHttpRequest object itself, which, if desired, can be queried for details about the status of the request.

The job performed by the load handler is simple: take the response data and dynamically set it into an element named `displayArea` using that element's `innerHTML` property.

The `displayArea` element is simply defined as an initially empty span element:

```
<div>
  Phone # is: <span id="displayArea"></span>
</div>
```

That pretty much sums up all the parts that need to go into our page. When displayed in a browser, it looks as shown in figure 4.1. The entire code for this rather Spartan-looking page is shown in listing 4.1.

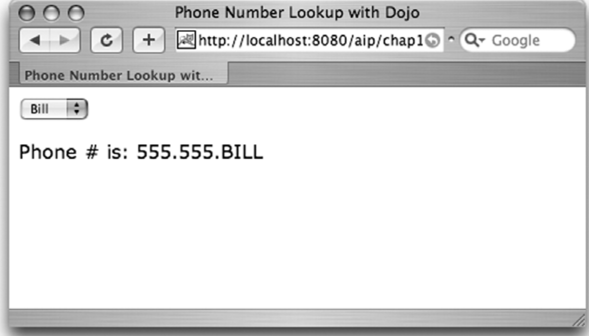

Figure 4.1
Phoning Mr. Bill

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 4.1 Phone number lookup with Dojo

```
<html>
  <head>
    <title>Phone Number Lookup with Dojo</title>
    <script type="text/javascript" src="dojo.js"></script>
    <script type="text/javascript">
      function lookup() {
        dojo.io.bind(
          {
            url: 'phone.jsp?who='+document.lookupForm.who.value,
            mimetype: 'text/plain',
            load:
              function(type, data, req) {
                document.getElementById('displayArea').
                  innerHTML=data;
              }
          }
        );
      }
    </script>
  </head>

  <body>
    <form name="lookupForm" onsubmit="return false;">
      <select name="who" onchange="lookup();">
        <option value="JOHN">John</option>
        <option value="MARY">Mary</option>
        <option value="BILL">Bill</option>
      </select>
    </form>
    <div>
      Phone # is: <span id="displayArea"></span>
    </div>
  </body>

</html>
```

Discussion

This section introduced us to using the `dojo.io.bind()` function to make Ajax requests to server-side resources.

A comparison of the amount of JavaScript code necessary to make such a request ourselves using the XHR object directly, versus using the Dojo function, would show a nontrivial, but hardly earth-shattering, reduction. However, more so than just lines of code, use of a toolkit such a Dojo helps reduce the *complexity* of the code by reducing the amount of “plumbing” code on the page so that the code

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

that remains is core code that is focused on providing the page's functionality. Such reduction in plumbing code, especially over the course of a large and complicated page, can significantly reduce the complexity of the page even if it makes only a small dent in the number of lines of code.

4.1.2 Automatic form marshaling with Dojo

The previous section showed how easy it was to use the Dojo `dojo.io.bind()` function to make a call back to the server to retrieve data asynchronously. One issue with that example was that in order to construct the URL to perform the GET operation, we needed to build a query string to append onto our URL. Generally, building query strings for URLs by hand is something that's best avoided in order to steer clear of common pitfalls such as

- Syntax issues, such as how many times you have used the ? in place of the &, and vice versa
- Incorrect encoding (or complete lack thereof) of the parameter names and values

But even if we don't want to have to deal with it, *something* has to build the query string.

Problem

We want to avoid having to build query strings to the URLs that we will pass to the Dojo `bind()` function. We think, "Wouldn't it be great if we could just 'submit' the form containing our select element to the asynchronous request instead of having to read the value and build the query string ourselves?"

As it happens, Dojo allows us to do just that.

Solution

The parameter object for the `dojo.io.bind()` function accepts a property named `formNode`, which lets us specify the DOM element node of a form element whose controls are passed as request parameters to the asynchronous request. This allows us to effectively "submit" the form via the Ajax request, even though we know that the form isn't actually being submitted.

This entails making only a few changes to the code of our previous section. First, we change the call to the `bind()` function as follows:

```
dojoio.bind(
  {
    url: 'phone.jsp',     <-(1) Look, Ma! No params!
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      mimetype: 'text/plain',
      load:
        function(type, data, req) {
          document.getElementById('displayArea').innerHTML = data;
        },
      formNode: document.lookupForm      ❷ Specifies form to submit
    }
  );
```

Note that a query string is no longer constructed and placed on the URL that we set as the value of the `url` property ❶, and that we have added a `formNode` property ❷ that specifies a reference to the form containing the select element. This causes the `bind()` function to automatically marshal the values of the control elements in the specified form and pass them as parameters to the asynchronous request. The changed page, with modifications highlighted in bold, is shown in listing 4.2.

Listing 4.2 Asynchronous form submission with Dojo

```
<html>
  <head>
    <title>Phone Number Lookup with Dojo</title>
    <script type="text/javascript" src="dojo.js"></script>
    <script type="text/javascript">
      function lookup() {
        dojo.io.bind(
          {
            url: 'phone.jsp',
            mimetype: 'text/plain',
            load:
              function(type, data, req) {
                document.getElementById('displayArea').innerHTML=data;
              },
            formNode: document.lookupForm
          }
        );
      }
    </script>
  </head>

  <body>
    <form name="lookupForm" onsubmit="return false;">
      <select name="who" onchange="lookup();">
        <option value="JOHN">John</option>
        <option value="MARY">Mary</option>
        <option value="BILL">Bill</option>
      </select>
    </form>
    <div>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        Phone # is: <span id="displayArea"/>
      </div>
    </body>

  </html>
```

Discussion

This example showed us a way to easily submit a form to an asynchronous request without the cumbersome requirement of building a query string from the values of the form controls. By merely passing a reference to the Dojo function, we ensure that the mechanics are handled on our behalf.

While gathering form values and constructing a query string for the URL is by no means rocket science, it's a rather messy and onerous task, and one in which it is easy to introduce silly errors. By taking care of this tiresome task, this aspect of Dojo provides a good example of how providing features that may not greatly reduce lines of code can still greatly reduce the complexity of the on-page code.

4.2 Prototype

Unless you skipped over the previous chapter, you've already been introduced to Prototype, a JavaScript toolkit that aims to make the life of DHTML coders easier. It's a popular toolkit that is not only useful in its own right but has also been used as the basis for other toolkits and frameworks such as Scriptaculous, Ruby on Rails, and Rico.

In order to use Prototype, all you need is the `prototype.js` file, which can be downloaded from the Prototype site thttp://prototype.conio.net/. Like the Dojo toolkit, Prototype offers a wide range of DHTML features. If you did breeze over the Prototype section of chapter 3, you might want to go back and read that section before continuing, as we'll be using some of the more useful Prototype extensions in our example code.

4.2.1 Asynchronous requests with Prototype

Like the Dojo toolkit, Prototype provides a number of easy ways to make asynchronous requests via Ajax. Let's start by looking at Prototype's means for making a basic request.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

On an order entry page, we are faced with the common problem of dynamically populating the contents of a dropdown control (`<select>` element) based on the selection made in another. For this example, we'll present a dropdown with a list of colors in which our collection of T-shirts are available. Based on the color selection, we need to dynamically consult our inventory database and populate the sizes dropdown to show only sizes that we actually have on hand for that color.

Solution

First, we import the Prototype library. Assuming that we have placed the `prototype.js` file in the same folder as the HTML page, that would be as easy as

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

Now, let's set up our form. For simplicity's sake, we'll only include the two dropdown elements and a submit button. Obviously, many other controls would be needed for an actual order form.

```
<form action="/submitOrder" name="tshirtForm">
  <label>T-shirt color:</label>
  <select name="color" id="color" onchange="updateSizes();">   <-- ❶ Defines onchange handler
    <option value="">Select color</option>
    <option value="cardinal">Cardinal</option>
    <option value="ecru">Ecru</option>
    <option value="hunter">Hunter</option>
    <option value="azure">Azure</option>
  </select>
  <label>Size:</label>
  <select name="size" id="size" disabled="disabled">   <-- ❷ Creates sizes dropdown
    <option value="">Select size</option>
  </select>
  <input type="submit"/>
</form>
```

A few things are notable about this form. First, note that the form's control elements have been given both an `id` and a `name` attribute, and that these values are the same. This allows us to refer to the control elements by either ID or by name, and since these identifiers exist in separate JavaScript namespaces, nothing will be confused by the fact that we used the same value for both. We also defined an `onchange` event handler on the colors dropdown ❶ so that we can react when the user selects a color from the list. The sizes dropdown ❷, which is empty except for a "Select size" directive, is initially disabled; it makes no sense for users to click on it until it's populated with some values.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When the user selects a color from the list of available colors, the `updateSizes()` function is invoked:

```
function updateSizes() {
  if ($F('color')=='') return;                          ❶ Obtains value of color dropdown
  new Ajax.Request('getSizes.jsp?color=' + $F('color'),
    {
      method: 'get',
      onSuccess: populateSizes,
      onFailure: function(r) {
        throw new Error( 'Fetch sizes failed: '        ❷ Throws error
          + r.statusText );
      }
    }
  );
}
```

The first thing that this handler does is use the `$F()` function to obtain the value of the color dropdown ❶ and exit if no color was actually selected (the user could click on the "Select color" entry, which we're just using as a helpful label). We could be more robust here in order to ensure that the size element is placed into a known state, but we're focusing on the Ajax request for now.

If that check passes, an asynchronous request is made using Prototype's Ajax.Request object. The asynchronous request itself is triggered by constructing a new instance of Ajax.Request, passing two parameters: the URL for the request, and a hash object containing properties that specify the options of the request. (We used this same technique in chapter 3, and it is also employed by the Dojo toolkit.)

The URL specifies a JSP file that we will use to simulate a database lookup into our inventory, and is passed the value of the chosen color. In the options parameters object, we specify the HTTP method as a GET (Prototype insists on lowercase here) with the `method` property, and provide function references for success and failure handlers with `onSuccess` and `onFailure`, respectively.

The failure handler, which is passed a reference to the XHR instance, throws an error depicting the failure status ❷. The success handler is a reference to the `populateSizes()` function.

To make life easy on the client-side code (it's almost always a good idea for the server-side code to "take one for the team" and handle as much of the complex processing as possible to help simplify the client-side code), the JSP will return as its response a JSON string containing the notation for a JavaScript array of the available sizes. A typical response might be

```
['Small','Medium','Large','XL','XXL']
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The code for the JSP, which utilizes the JSTL (JSP Standard Tag Library) core actions, is

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:choose>
  <c:when test="${param.color =='azure'}">
    ['Small','Medium','XXXL']
  </c:when>
  <c:when test="${param.color =='cardinal'}">
    ['Medium','Large','XL']
  </c:when>
  <c:when test="${param.color =='ecru'}">
    ['Small','Medium','Large','XL','XXL','XXXL']
  </c:when>
  <c:when test="${param.color =='hunter'}">
    ['Small','Medium','Large','XL','XXL']
  </c:when>
</c:choose>
```

Of course, in a real-world situation, this would be a servlet or other server-side resource that would perform a database lookup rather than returning hard-coded values.

When this JSP returns its response, the Prototype Ajax.Request object will invoke the `populateSizes()` function (assuming all has gone well, of course), which we have defined as follows:

```
function populateSizes(r) {
  eval('var sizes=' + r.responseText);
  var sizeElement = $('size');                      // ❶ Obtains dropdown element reference
  while (sizeElement.options.length > 1) {          // Empties dropdown element
    sizeElement.remove(1);
  }
  for (var n = 0; n < sizes.length; n++) {
    sizeElement.add(
      new Option(sizes[n],sizes[n]),document.all ? 0 : null    // ❷ Adds the options
    );
  }
  sizeElement.disabled = false;
}
```

This handler is passed a reference to the XHR instance, and the first thing that it does is to obtain the results of the response. Using the JavaScript `eval()` function, we evaluate the JSON response text and assign it to a variable for use later in the function. After the evaluation, this variable will contain a reference to a JavaScript string array, specifying the size values that our JSP returned for the passed color.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We want to add those values to the sizes dropdown, so we obtain a reference to the dropdown element ❶ and empty it by removing all of its options with the exception of the first (which contains our helpful "Select size" label) ❷. We then iterate over each of our returned size values and add a new option containing the size to the sizes dropdown.

When we make the call to the select element's `add()` method ❷, the second parameter warrants some explanation:

```
document.all ? 0 : null
```

The W3C specification for the `add()` method of the select element calls for the second parameter to specify the index of the existing option before which the new option will be inserted, or `null` in order to insert the new option at the end of the list.

Internet Explorer, however, insists on using a zero rather than `null` to indicate that the new option be placed at the end, so we do a little browser detection and provide the appropriate value. Usually *object detection* rather than browser detection is recommended for making client-dependent choices, but in this case there's no object to test in order to make the appropriate decision.

Finally, after all the sizes have been added, the sizes dropdown is enabled. In a browser, before a color selection, we would see a page such as shown in figure 4.2. The same page after a color selection is shown in figure 4.3. The completed code for our page is shown in listing 4.3.

Figure 4.2
Color selection page: before

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

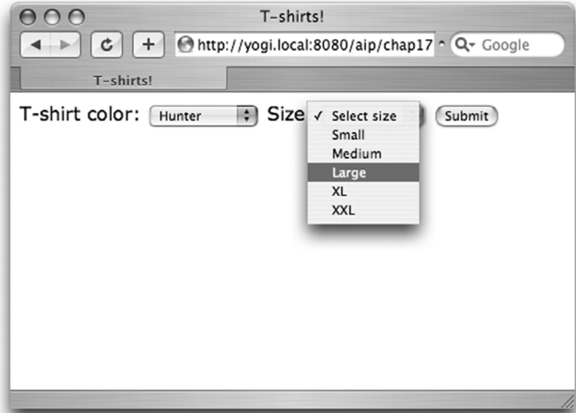

Figure 4.3
Color selection page: after

Listing 4.3 Dynamic lookup with Prototype

```
<html>
  <head>
    <title>T-shirts!</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script>
      function updateSizes() {
        if ($F('color')=='') return;
        new Ajax.Request('getSizes.jsp?color=' + $F('color'),
          {
            method: 'get',
            onSuccess: populateSizes,
            onFailure: function(r) {
              throw new Error( 'Updates sizes failed: ' +
                                 r.statusText );
            }
          }
        );
      }

      function populateSizes(r) {
        eval('var sizes=' + r.responseText);
        var sizeElement = $('size');
        while (sizeElement.options.length > 1) sizeElement.remove(1);
        for (var n = 0; n < sizes.length; n++) {
          sizeElement.add(
            new Option(sizes[n],sizes[n]), document.all ? 0 : null
          );
        }
        sizeElement.disabled = false;
      }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    </script>
  </head>

  <body>
    <form action="/submitOrder" name="tshirtForm">
      <label>T-shirt color: </label>
      <select name="color" id="color" onchange="updateSizes();">
        <option value="">Select color</option>
        <option value="cardinal">Cardinal</option>
        <option value="ecru">Ecru</option>
        <option value="hunter">Hunter</option>
        <option value="azure">Azure</option>
      </select>
      <label>Size:</label>
      <select name="size" id="size" disabled="disabled">
        <option value="">Select size</option>
      </select>
      <input type="submit"/>
    </form>
  </body>
</html>
```

Discussion

Like the Dojo toolkit, Prototype allows us to make asynchronous Ajax requests in a simpler fashion (compared with using the XHR object directly) by handling the details of making the request and handling the state change callback. This allows us to abstract the code necessary to initiate and handle the request, focusing on the processing at hand.

The value of the small level of abstraction that Prototype provides in this area may not be apparent in an example of this size, but on a more complicated real-world page, the advantages of keeping the code neat can become a major factor in the maintainability and extensibility of the page.

4.2.2 Automatic updating with Prototype

In the previous problem, the data that came back to our page from the server required some processing in order to display; a string array of values needed to be converted to option elements with which to populate a select element.

But frequently, we may want to display data that we get back from the server as is with no interpretation necessary. For just such occasions, Prototype provides the Ajax.Updater class to make this process easy and painless.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

Upon some event on the page—a button click, perhaps—we wish to obtain the date and time from the server and display it on the page.

Solution

As expected, we start by importing the Prototype toolkit:

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

We'll be triggering the update of the date and time information as a result of a button press, so we define a button:

```
<button type="button" onclick="update();">Click me!</button>
```

and an initially empty container in which to display the date and time:

```
<span id="timeDisplay"></span>
```

The `onclick` event handler of the button triggers the `update()` function in which we employ the Ajax.Updater object:

```
function update() {
  new Ajax.Updater('timeDisplay','date.jsp',
    {
      method: 'get'
    }
  );
}
```

As we did with the Ajax.Request object, we use the Ajax.Updater object by creating an instance and passing the relevant information in the constructor's parameters.

The first parameter specifies the `id` of the element into which the response text will be placed, and the second parameter specifies the URL from which to obtain the response. In this case the server-side resource is a JSP page named `date.jsp`, which employs the JSTL internationalization actions to format and return the current time:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<jsp:useBean id="now" class="java.util.Date"/>
<fmt:formatDate value="${now}" pattern="MMMM dd, yyyy hh:mm aa"/>
```

The third parameter is a JavaScript object containing the request options, just as we saw with Ajax.Request. In this case, we merely specify the HTTP method as GET.

This results in the display (after a click of the button) shown in figure 4.4. The complete code that resulted in this page is shown in listing 4.4.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

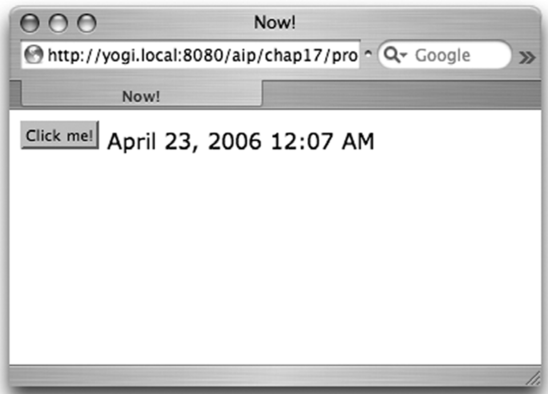

Figure 4.4 Finding out what time it is with Prototype

Listing 4.4 Keeping up-to-date

```
<html>
  <head>
    <title>Now!</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script>
      function update() {
        new Ajax.Updater( 'timeDisplay', 'date.jsp',
          {
            method: 'get'
          }
        );
      }
    </script>
  </head>

  <body>
    <button type="button" onclick="update();">Click me!</button>
    <span id="timeDisplay"></span>
  </body>
</html>
```

Discussion

Updating an HTML element with data gathered from an Ajax request is an extremely common occurrence on many Ajax-enabled pages. By providing the Ajax.Updater object, Prototype reduces the amount of code necessary to perform this common task to the barest minimum.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This is especially useful for requests made to server-side resources that can easily return already-formatted HTML code, such as JSP pages, PHP scripts, or even static HTML files. Having the server-side resource perform the formatting can greatly reduce the amount of processing necessary on the page to gather raw data and use it to either format HTML strings for use with `innerHTML`, or to use the DOM API to dynamically build the desired HTML elements.

4.2.3 Periodic updating with Prototype

In the previous problem, we used the Prototype Ajax.Updater object to cause an HTML element container to be updated automatically with preformatted response data from the server; we used the current date and time as an example.

We were able to do this with a minimum of code, and it was convenient to just be able to name the target display element by its `id`. There are times, however, where we would want to execute the same function, but at set intervals in order to ensure that the displayed data will be as up-to-date as feasible.

Problem

We wish to obtain and display data from the server (using the current date and time once again) at periodic intervals.

Solution

Starting with the code of the previous solution, we could easily achieve our goal by invoking the services of the Ajax.Updater class in a timeout handler invoked by using the JavaScript `window.setTimeout()` or `window.setInterval()` function. But once again, Prototype makes things even easier for us by providing the Ajax.PeriodicalUpdater class. To use this class, we need only make a few minor adjustments to our previous solution.

We'll be calling a slightly different back-end processing resource, `date2.jsp`, which returns as its response the current data and time in a manner similar to its predecessor. But this time we include the seconds in the time value so that we can detect updates on a second-by-second basis:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<jsp:useBean id="now" class="java.util.Date"/>
<fmt:formatDate value="${now}" pattern="MMMM dd, yyyy hh:mm:ss aa"/>
```

The changes to the `update()` function include using the Ajax.PeriodicalUpdater class and specifying the interval at which the automatic updates are to occur:

```
function update() {
  new Ajax.PeriodicalUpdater('timeDisplay','date2.jsp',
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      {
        method: 'get',
        frequency: 5
      }
    );
  }
```

As you can see, only two changes are involved. First, we create an instance of Ajax.PeriodicalUpdater that refers to the new JSP page in the request URL. Second, we introduce a `frequency` property in the parameters object that specifies the number of seconds between updates.

When we display this page in the browser, it looks and acts exactly like our previous solution, except that once the button is clicked to display the date, it will automatically update every 5 seconds. The complete code for this page, with the changes from the previous solution highlighted in bold, is shown in listing 4.5.

Listing 4.5 Keeping constantly up-to-date with Prototype

```
<html>
  <head>
    <title>Right now!</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script>
      function update() {
        new Ajax.PeriodicalUpdater( 'timeDisplay', 'date2.jsp',
          {
            method: 'get',
            frequency: 5
          }
        );
      }
    </script>
  </head>

  <body>
    <button type="button" onclick="update();">Click me!</button>
    <span id="timeDisplay"/>
  </body>

</html>
```

Discussion

With very little effort, we've updated our previous example with the ability to automatically keep the display of the server-provided data up-to-date at a frequency that best suits our page. The Prototype Ajax.PeriodicalUpdater class made it easy

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

for us to schedule the automatic updates without having to code any scheduling logic or write timeout handlers.

When using such automatically fired updaters, care should be taken not to overwhelm the server with requests. Although initially it may seem like a good idea to fire off an update every second in order to keep the display as up-to-date as possible, imagine the load that could be created when hundreds, or even thousands, of visitors to our site are all looking at a page that hammers the server with requests for updated data. Granted, the request and the generated response are usually small (and certainly are in this example), but it's possible to kill the server via the proverbial "death by a million paper cuts."

When choosing what to automatically update and at what interval, the expected number of simultaneous visitors, the size of the update response, the process required to generate that response, and the load-handling capability of the server configuration all need to be taken into account. At best, the update interval could be factored out into a configuration file rather than being hard-coded into the pages, so that an administrator could adjust the value on the fly without having to recode pages.

4.3 jQuery

jQuery, a self-professed "new type" of JavaScript library, operates from a slightly different viewpoint than the toolkits we've seen so far in this chapter. It purports to change the way that you write JavaScript, and quite truly, adopting the jQuery philosophy can make a huge impact on how you develop the script for your pages.

The jQuery downloads and documentation can be found at http://jquery.com/. You can download this JavaScript library as either an uncompressed library (with human-readable code) or a smaller compressed file (not readable).

In either case, import the jQuery script file into any pages on which you wish to use jQuery. For the purposes of this section, we'll use the uncompressed (readable) version of the script file, place it in the same folder as our example pages (for easy importing), and name it `jquery.js`.

4.3.1 jQuery Basics

Before diving into making Ajax requests with jQuery, let's take a look at some of the basic concepts that we need to have under our belts before beginning to make sense of how jQuery operates.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This section will by no means be a complete primer on jQuery—that would take much more space than we have allotted here—but it should give you an idea of the philosophy behind jQuery's modus operandi.

The jQuery wrapper

Other libraries that we have seen, particularly Prototype, operate by introducing new classes and by extending the built-in JavaScript classes in order to augment the capabilities of the script on our pages. In chapter 3, for example, we saw how Prototype extended the Object, Function, and Array classes.

jQuery takes a different approach.

Rather than extending classes, jQuery provides a new class, appropriately named *jQuery*, that serves as a *wrapper* around other objects in order to provide extended operations upon those objects. The concept of a wrapper object is not foreign to advanced developers of object-oriented programs. This pattern is often used as an *adapter* to present an interface for manipulating an object that is different from the original object's interface.

In jQuery, most operations are performed by using the jQuery wrapper around a set of items and calling wrapper methods that operate upon the wrapped items. In order to make expressions and statements containing jQuery wrappers terser, the jQuery class is mapped to `$`. This is not to be confused with Prototype's use of `$()`, which serves a completely different purpose.

The jQuery object can wrap a number of different object types, and what it can do for us depends on what has been wrapped. For example, we can wrap an HTML snippet:

```
$("<p>What's cooking?</p>")
```

This constructs a DOM fragment from the HTML that we can then operate upon with jQuery's methods. For example, if we wanted to append this fragment to the end of the document, we could use

```
$("<p>What's cooking?</p>").appendTo("body");
```

As Ajax developers who often have a need to generate new DOM elements, the advantages of this convenient and short means to effect such additions should be readily apparent.

In addition to adding new DOM elements, we often find ourselves needing to manipulate existing elements in our pages. The jQuery wrapper also allows us to wrap existing elements by passing a string to the `$()` wrapper that provides a number of ways to identify the items to be wrapped: CSS selectors,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

XPath expressions, and element names. We'll be using CSS selectors a great deal within our example code. Consider the following:

```
$("div")
```

This will cause all `<div>` elements in the document to be wrapped for manipulation. Another example is

```
$("#someId")
```

This wraps the DOM element with the id of `someId` for manipulation. Here's yet a third example:

```
$(".someClass")
```

This will wrap all elements, regardless of type, that possess the CSS class name of `someClass`.

The authors of jQuery were very clever in using CSS selectors and XPath to identify target elements as opposed to inventing some jQuery-specific syntax that users of jQuery would be forced to adopt. By using mechanisms that we, as page developers, are already familiar with, they have made it far easier for us to adopt and use jQuery to identify the elements that we wish to manipulate.

It is also possible to wrap other items such as elements and functions. We'll be seeing examples later in this section.

Chaining jQuery operations

jQuery sensibly allows us to string together numerous operations into a single expression. Most of the jQuery wrapper methods return a reference to the jQuery wrapper object itself so that we can just keep tacking operations onto a single expression when we need to perform multiple manipulations on the wrapped object(s).

Consider the case where we might want to add a CSS class to an element (whose `id` is `something`) and then cause it to be shown (assuming it was initially hidden). Rather than

```
$('#something').addClass('someClass');
$('#something').show();
```

we would write

```
$('#something').addClass('someClass').show();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Executing code when the document is ready

Frequently on our pages, we need some initialization code to execute in order to prepare the page before the user gets a chance to interact with it. Generally we use the window's `onload` event handler for such initializations. This guarantees that the page has completed loading prior to executing the `onload` code, thereby guaranteeing that the DOM elements exist and are ready for manipulation.

But one problem with relying on `onload` is that not only does it wait until the document body has been loaded, but it also waits for images to load. Since images must be fetched from the server if the browser has not cached them, this can sometimes extend the point at which the initialization code runs far beyond the point at which the document has been loaded and the code is safe to execute.

jQuery solves this problem for us by introducing the concept of the "document ready handler." This mechanism causes a function to execute when the document has loaded but prior to waiting for any images and the `onload` event handler.

The syntax for employing this mechanism is to wrap the document element and to call the `ready()` method on the wrapped document:

```
$(document).ready(function);
```

Whatever function is passed to `ready()` will execute when the DOM is ready for manipulation. Note that when you use both the ready mechanism and an `onload` event handler on a page, both handlers will execute, with the `ready` event handler triggered prior to the `onload` event handler.

A shorthand notation for a `ready()` handler can be used by wrapping a function in the jQuery wrapper. The code fragment

```
$(function);
```

is equivalent to the code fragment for declaring a `ready()` handler that was presented earlier.

Using jQuery and Prototype together

Prototype is a very popular library, and jQuery is rapidly gaining ground. As such, it's not unlikely that page authors might wish to use the power of both libraries on the same page.

In general, jQuery follows best-practice guidelines and avoids polluting the global namespace—for example, by placing such constructs as utility functions within the jQuery namespace. But one area of conflict, which we've already alluded to earlier, is the use of the `$` as a global name.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

jQuery, being a good library citizen, has anticipated this issue. When using Prototype and jQuery on the same page, calling the jQuery utility function `jQuery.noConflict()` any time after both libraries have been loaded will cause the functionality of the `$` name to revert to Prototype's definition.

jQuery functionality will still be available through the `jQuery` namespace, or you could define your own shorthand alias. For those times when you use jQuery together with Prototype, the jQuery documentation suggests the following alias:

```
var $j = jQuery;
```

That's enough preliminaries!

We'll see more use of jQuery methods within the solutions in this section. But even so, we'll only be lightly touching on jQuery's capabilities. If after reading these solutions you find yourself intrigued by jQuery's capabilities, we strongly urge you to visit http://docs.jquery.com/ to read the extensive online documentation and find out what other capabilities jQuery has to offer.

4.3.2 Asynchronous loading with jQuery

jQuery provides a fairly large number of methods to make Ajax requests. Some are simple and useful high-level methods that initiate Ajax requests to perform some of the most commonly required tasks. Others are more low-level, providing control over every aspect of the Ajax request.

We'll employ a representative handful of these methods in the solutions within this section. First, let's tackle one of the most common of Ajax interactions: obtaining dynamic content from the server.

Problem

Let's imagine that we own an *eFridge*—a hypothetical high-tech refrigerator that not only keeps track of what its contents are, but also provides an Internet interface that server software can use to communicate and interact with the eFridge.

The imaginary technology used by the eFridge to keep track of its inventory is unimportant. It could be bar-code scanning, RFID (Radio Frequency Identification) tags, or some yet-to-be-imagined technology. All we care about as page authors is that we have a server component to which we can make requests in order to obtain information about the state of our food!

The page we'll focus on will present a list of the items that are in our eFridge. Upon clicking on an item in this list, more information about the item will be displayed.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

For this problem, we'll assume that the page was prepopulated with the list of items by whatever server-side templating mechanism generated our page. In the next section, we'll see a technique to obtain this list dynamically from the server.

Solution

To begin, in order to use jQuery on a page it is necessary to import the jQuery library:

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

The list of items in the eFridge, which we're assuming was generated on our behalf by some server-side mechanism, is presented in a select element:

```
<form>
  <select id="itemsControl" name="items" size="10">
    <option value="1">Milk</option>
    <option value="2">Cole Slaw</option>
    <option value="3">BBQ Sauce</option>
    <option value="4">Lunch Meat</option>
    <option value="5">Mustard</option>
    <option value="6">Hot Sauce</option>
    <option value="7">Cheese</option>
    <option value="8">Iced Tea</option>
  </select>
</form>
```

For the purpose of this example, we're only showing eight items. The average refrigerator would probably contain more than this, but we all know fast-food junkies whose refrigerator contents are sometimes pretty sparse.

The server (perhaps some "eFridge driver") assigns each item an identification number that is used to uniquely identify each item—in this case, a simple sequential integer value. This identifier is set as the `value` for each `<option>` representing an item.

Even though we know that we need the select control to react to user input, note that no handlers are declared within the markup that creates the `<select>` element. This brings up another philosophy behind the design of jQuery.

One of the goals of jQuery is to make it easy for page authors to separate script from document markup, much in the same manner that CSS allows us to separate presentation from the document markup. Granted, we could do it ourselves without jQuery's help——after all, jQuery is written in JavaScript and doesn't do anything we couldn't do—but jQuery does a lot of the work for us, and is designed with the goal of easily separating script from document markup. So, rather than

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

adding an `onchange` event handler directly in the markup of the `<select>` element, we'll use jQuery's help to add it under script control.

We can't manipulate the DOM elements on our page until after the document is ready, so in the `<script>` element in our page header, we'll institute a jQuery `ready()` handler as we previously discussed. Within that handler, we'll use jQuery's method to add a `change` handler to an element, as shown in the following code fragment:

```
$(document).ready(function(){
   $('#itemsControl').change(showItemInfo);
});
```

In the `ready()` handler, we create a jQuery instance that wraps the `<select>` element, which we have given the `id` of `itemsControl`. We then use the jQuery `change()` method, which assigns its parameter as the change handler for the wrapped element.

In this case, we've identified a function named `showItemInfo()`. It's within this function that we'll make the Ajax request for the item that is selected from the list:

```
function showItemInfo() {
   $('div#itemData').load(          ❶ Wraps element and invokes load method
     'fetchItemData.jsp',           ❷ Identifies server-side resource
     {itemId: $(this).val()}        ❸ Obtains item id and passes as parameter
   );
}
```

jQuery provides a fair number of different ways to make Ajax requests to the server. For the purposes of this solution, we'd like to fetch a pre-formatted snippet of HTML from the server (containing the item data) and load it into a waiting element, that is, a `<div>` element having an `id` of `itemData`. The jQuery `load()` method ❶ serves this requirement perfectly.

This method fetches a response from a URL provided as its first parameter and inserts it into the wrapped DOM element. A second parameter to this function allows us to pass an object whose properties serve as the parameters for the request. A third parameter can be used to specify a callback function to be executed when the request completes.

First, we wrap a DOM element ❶ identified by the CSS selector `div#itemData`, which is an empty `<div>` element into which we want the item data to be loaded. Then, using the `load()` method, we provide the URL to a JSP page ❷ that will fetch the item data identified by the `itemId` request parameter supplied in the second method parameter ❸.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Figure 4.5 Got milk?

The value of that parameter needs to be the value of the option that the user clicks on in the `<select>` element. Because the `<select>` element is set as the function context of the change handler, it is available to it via the `this` reference. We wrap that reference and use JQuery's `val()` method to obtain the current selected value of the control ❸.

Since all we want to do is to load the item data into the DOM, we have no need for a callback and omit the third parameter to the `load()` method.

That's all there is to it.

jQuery's capabilities have taken a very common procedure that might have taken a nontrivial amount of code to implement and allow us to perform it with very few lines of simple code. The JSP page that gets invoked by this handler uses the value of the `itemId` request parameter to fetch the info for the corresponding item and formats it as HTML to be displayed on the page.

Our finished page, shown in figure 4.5 after selecting a refrigerator item, is laid out in its entirety in listing 4.6.

Listing 4.6 What's for dinner with jQuery

```
<html>
  <head>
    <title>What's for dinner?</title>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function(){
         $('#itemsControl').change(showItemInfo);
      });
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        function showItemInfo() {
          $('div#itemData').load(
            'fetchItemData.jsp',
            {itemId: $(this).val()}
          );
        }
      </script>
      <style type="text/css">
        form,#itemData {
          float: left;
        }
      </style>
    </head>

    <body>
      <form>
        <select id="itemsControl" name="items" size="10">
          <option value="1">Milk</option>
          <option value="2">Cole Slaw</option>
          <option value="3">BBQ Sauce</option>
          <option value="4">Lunch Meat</option>
          <option value="5">Mustard</option>
          <option value="6">Hot Sauce</option>
          <option value="7">Cheese</option>
          <option value="8">Iced Tea</option>
        </select>
      </form>

      <div id="itemData"></div>
    </body>
  </html>
```

Discussion

This section introduced us to one of JQuery's means of performing Ajax requests, the `load()` method.

The jQuery `load()` method is very well suited for use with server-side templating languages such as JSP and PHP that make it a snap to format and return HTML as the response. The `fetchItemData.jsp` file, as well as the Java classes that fake the eFridge functionality, are available in the downloadable source code for this chapter.

A few other important jQuery features are also exposed in this solution. For example, we used a `ready()` handler to trigger the execution of code that must execute before a user is allowed to interact with the page, but after the entire DOM has been constructed for the page.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We also saw the `val()` method, which returns the value of the wrapped input element. If more than one element is wrapped, the value of the first matched element is returned by this method.

In this solution, we assumed that the original list of eFridge contents was generated by whatever server-side resource produced the page—a JSP template, for example. That's a common expectation for a web application, but in the interest of exploring more of jQuery's Ajax abilities, let's pretend that we need to fetch that list dynamically upon page load in the next problem.

4.3.3 Fetching dynamic data with jQuery

In the previous section we were introduced to the jQuery `load()` method, which made it extremely easy to perform the common task of fetching an HTML snippet to load into a DOM element. While the utility of this method cannot be dismissed, there are times when we might want to exert more control over the Ajax request process, or to obtain data (as opposed to preformatted HTML) from the server.

In this section we'll explore more of what jQuery has to offer in the Ajax arena.

Problem

We wish to augment the code of the previous section to obtain the list of items in the eFridge from a page-initiated asynchronous request.

Solution

Reviewing the previous solution, we can readily see that in order to load the select options dynamically, the changes that we would need to make are to remove the `<option>` elements from the `<select>` element and to add code to the `ready()` handler to fetch and load the items. But before we embark upon that effort, we're going to change the way that we coded the `showItemInfo()` handler function if for no other reason than as an excuse to further explore jQuery's capabilities. Rather than using the `load()` method of the jQuery wrapper, we're going to use one of jQuery's utility functions: `$.get()`.

Hey, wait a minute! What's that period character doing in there? That's not the `$()` wrapper that we've been using up to now!

Not only does jQuery provide the wrapper class that we've make good use of up to this point, but it also provides a number of *utility functions,* many implemented as class methods of the `$` wrapper class.

If the notation `$.functionName()` looks odd to you, imagine the expression without using the `$` alias for the jQuery function:

```
jQuery.get();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

OK, that looks more familiar. The `$.get()` function is defined as a class method—that is, a function property of the jQuery wrapper function. (If the concept of a class method is still giving you a headache, you might wish to review section 3.1.3.) Although we know them to be class methods on the jQuery wrapper class, jQuery terms these methods *utility functions* and to be consistent with the jQuery terminology, that's how we'll refer to them in this section.

The `$.get()` utility function accepts the same parameters as the `load()` method: the URL of the request, a hash of the request parameters, and a callback function to execute upon completion of the request. When using this utility function, because there is no object being wrapped that will automatically be injected with the response, the callback function, although an optional parameter, is almost always specified. It is the primary means for causing something to happen when the request completes.

It should also be noted that the callback function can be specified as the *second* parameter to this utility function when no request parameters need to be passed. Internally, jQuery uses some JavaScript sleight of hand to ensure that the parameters are interpreted correctly.

The rewritten `showItemInfo()` handler using this utility function is as follows:

```
function showItemInfo() {
  $.get('fetchItemData.jsp',
        {itemId: $(this).val()},
        function(data) {
          $('#itemData').empty().append(data);
        }
  );
}
```

Aside from using the `$.get()` utility function, another change to the code of the previous solution was the addition of a callback function as the third parameter, which we use to insert the returned HTML into the `itemData` element.

In doing so, we make use of two more wrapper methods: `empty()`, which clears out the wrapped DOM element, and `append()`, which adds to the wrapped element the HTML snippet passed to the callback in the `data` parameter.

Now we're ready to tackle loading the `<options>` from data that we will obtain from the server when the document is loading. In this case, we're going to obtain the raw data for the options from the server in the form of a JavaScript hash object. We could return the data as XML, but we'll opt to use JSON, which is easier for JavaScript code to digest.

jQuery comes to our rescue once again with a utility function that is well suited to this common task: the `$.getJSON()` utility function. This function accepts the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

now-familiar trio of parameters: a URL, a hash of request parameters, and a callback function.

The advantage that the `$.getJSON()` utility function brings to the table is that the callback function will be invoked with the already-evaluated JSON structure. We won't have to perform any evaluation of the returned response. How handy!

Using this utility method, the following line gets added to the document's `ready()` handler:

```
$.getJSON('fetchItemList.jsp',loadItems);
```

A JSP page named `fetchItemList.jsp` is used as the URL, and a function named `loadItems()` (whose definition we'll be looking at next) is supplied as the callback function. Note that, since we don't need to pass any request parameters, we can simply omit the object hash and provide the callback as the second parameter.

The `loadItems()`function is defined as

```
function loadItems(itemList) {                              <- 1 Invokes callback with evaluated JSON structure
  if (!itemList) return;
  for(var n = 0; n < itemList.length; n++) {
    $('#itemsControl').get(0).add(                          <- Locates select element
      new Option(itemList[n].name,itemList[n].id),          <- 2 Adds new option
                 document.all ? 0 : null
    );
  }
}
```

Recall that the `$.getJSON()` utility function invokes the callback with the JSON response already evaluated as its JavaScript equivalent ❶. In our solution, the `fetchItemList.jsp` page will return a response that contains

```
[
{id:'3',name:'BBQ Sauce'},
{id:'5',name:'Mustard'},
{id:'7',name:'Cheese'},
{id:'2',name:'Cole Slaw'},
{id:'4',name:'Lunch Meat'},
{id:'8',name:'Iced Tea'},
{id:'6',name:'Hot Sauce'},
{id:'1',name:'Milk'}
]
```

When our callback is invoked, this response string will already have been converted to an array of JavaScript objects, each of which contains an `id` and a `name` property, courtesy of jQuery. Each of these objects will be used to construct a new `<option>` element to be added to the select control ❷ in a similar fashion as we saw in the solution of section 4.2.1.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In order to add an option to the `<select>` element, we need a reference to that control's DOM element. We could just use `document.getElementById()` or `$()`, but we have chosen to do it the jQuery way with the `get()` wrapper method:

```
$('#itemsControl').get(0)
```

This method, when passed no parameters, returns an array of all the elements matched by the CSS selector of the jQuery wrapper on which it is invoked. If we only want one of those matches, we can specify a zero-based index as a parameter. In our case, we know that there will only be a single match to the selector because we used an `id`, so we specify an index of 0 to return the first matched element.

The code for the entire page, with changes from the previous solution highlighted in bold, is shown in listing 4.7.

Listing 4.7 More dinner with jQuery

```
<html>
  <head>
    <title>What's for dinner?</title>
    <script type="text/javascript" src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function(){
        $.getJSON('fetchItemList.jsp',loadItems);
        $('#itemsControl').change(showItemInfo);
      });

      function loadItems(itemList) {
        if (!itemList) return;
        for(var n = 0; n < itemList.length; n++) {
          $('#itemsControl').get(0).add(
            new Option(itemList[n].name,itemList[n].id),
                       document.all ? 0 : null
          );
        }
      }

      function showItemInfo() {
        $.get('fetchItemData.jsp',
              {itemId: $(this).val()},
              function(data) {
                $('#itemData').empty().append(data);
              }
        );
      }
    </script>
  </head>

  <body>
    <form style="float:left">
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      <select id="itemsControl" name="items" size="10">
      </select>
    </form>
    <div id="itemData" style="float:left"></div>
  </body>
</html>
```

Discussion

This section exposed us to more of jQuery's abilities in the areas of DOM manipulation and traversal, as well as Ajax request initiation. We saw the jQuery `$.get()` utility function, which made it easy for us to make Ajax requests using the HTTP GET method. A corresponding utility function named `$.post()` with the exact same function signature makes it equally easy to submit POST requests via Ajax. As both utility functions use the same parameter signature—most notably the request parameter hash—we can easily switch between which HTTP method we'd like to use without having to get bogged down in the details of whether the request parameters need to be encoded in the query string (for GET) or as the body of the request (for POST).

Another Ajax utility function, `$.getJSON()`, makes it incredibly easy for us to use the power of the server to format and return JSON notation. The callback for this operation is invoked with the JSON string already evaluated, preventing us from having to work with the vagaries of the JavaScript `eval()` function ourselves.

For occasions where we might wish to exert more control over, and visibility into, an Ajax request, jQuery provides a versatile utility function named `$.ajax()`. The online documentation provides more details about how to use this low-level utility function.

We also saw a handful of the powerful DOM manipulation wrapper methods such as `get()`, `empty()`, `val()`, and `append()`, all geared toward making it easy for us—as Ajax page developers—to manipulate the page DOM.

This is all just barely plumbing the depth of jQuery capabilities. For example, space prevents us from exploring the effects API, which provides fading, sliding, flashing, hovering. and even the ability to provide your own animations. You are urged to visit http://jquery.com/ for more information on jQuery and how it can help you write powerful Ajax applications.

Additionally, advanced developers might be interested in jQuery's plug-in API. This API is one of jQuery's most powerful assets, as anyone can extend the toolkit in a snap. For more information, please see http://docs.jquery.com/Plugins/Authoring.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

And now, as they say, for something completely different. Let's take a look at a fourth framework that approaches the issue of asynchronous requests from a new angle.

4.4 DWR

DWR stands for *Direct Web Remoting* and is a means of performing remote procedure calling from client-side JavaScript to server-side Java code using Ajax as the transport mechanism. In the libraries we've looked at so far in this chapter, the paradigm of submitting a request to a server-side resource and receiving a response has been maintained. With DWR, things are a bit different.

Remote Procedure Calling (RPC) is a mechanism to allow local code to call methods on objects that exist on a remote server as if that remote object were also local. Generally, RPC works (with a lot of hand-waving and glossing over details that aren't all that important to us at this point) by creating a local proxy interface that mimics the signature of the remote method (usually called a *stub*). The local code makes a call to the local interface, and an RPC agent on the local system marshals any input data for the call and performs the network processing necessary to pass that information to its counterpart running on the remote system. The remote agent unpacks the data into the appropriate formats and makes the call to the actual remote method. When the method returns, any returned data is marshaled and sent back to the local agent, which in turn returns control from the proxy stub to the original caller of the remote method.

From the point of view of the local calling code, the fact that all that data marshaling and network communications went on behind the façade of the proxy is hidden. Likewise, the remote method has no knowledge that it was invoked remotely.

DWR isn't exactly a pure RPC implementation for two main reasons:

- The signature of the proxy method used from JavaScript is not identical to that of the remote Java method.
- The call is not synchronous. Like other Ajax mechanisms, the invocation of the remote method is asynchronous and a handler function is invoked upon completion of the remote method.

But beyond any academic arguments over the purity of the implementation, DWR offers a clever means for those who prefer to think in terms of method calls rather than the traditional HTTP request-response cycle.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The DWR JAR file is available at http://getahead.ltd.uk/dwr/. After downloading, place this JAR file in the `WEB-INF/lib` folder of your web application.

4.4.1 Direct Web Remoting with DWR

DWR is cleverly implemented as a servlet that handles all the needs of the client. Most RPC implementations require the use of a preprocessor that creates the client and server stubs. DWR, on the other hand, generates the client-side stubs dynamically via a reference to the servlet that looks like a normal JavaScript file reference.

But before we get to that, some setup is required: the DWR servlet needs to be declared and mapped in the web application's deployment descriptor (`web.xml`).

To declare the servlet, add a `<servlet>` element to the deployment descriptor as follows:

```
<servlet>
  <servlet-name>DwrServlet</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
     <param-name>debug</param-name>
     <param-value>true</param-value>
  </init-param>
</servlet>
```

The `debug` init parameter is optional but is very useful during development. Don't include it for actual deployment because it will allow visitors with less-than-noble intentions to obtain information about your code that is best not shared. We'll see what this parameter does for us in just a little bit.

The servlet is mapped to a URL via

```
<servlet-mapping>
  <servlet-name>DwrServlet</servlet-name>
  <url-pattern>/dwrserver/*</url-pattern>
</servlet-mapping>
```

This maps any URL for your web application beginning with `/dwrserver` to the DWR servlet.

There's one more setup step, but it is dependent on how we are going to use DWR. So, on to a specific example...

Problem

On a page that contains a form with customer information, we want to automatically fill in the address information for a customer if the name of that customer can be uniquely found in our database. And, of course, we wish to do so asynchronously without the need for a page reload.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Solution

The first thing we'll do is set up a Java class that we can query for customer information, given a first and last name. Obviously, a real-world implementation of such a class would perform a database lookup, but for the purposes of this example, we'll just hard-code a false implementation behind the API for the class—one that assumes we have a single customer named Bill Moody.

```
public class CustomerFactory {

  public Customer findByName(String firstName,String lastName) {
    if ("Bill".equalsIgnoreCase(firstName) &&
        "Moody".equalsIgnoreCase(lastName)) {
      return new Customer("Bill", "Moody", "123 Nowhere Lane",
                          "Austin", "TX", "USA", "78701");
    } else {
      return null;
    }
  }
}
```

The `findByName()` method returns either an instance of a located customer, or `null` if none is found. In a real implementation it would also return `null` if multiple customers with the same name were found since it would not have enough information to uniquely identify a particular customer.

To define this class to the DWR engine, we create an XML file named `dwr.xml` in the `WEB-INF` folder:

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
  <allow>
    <convert converter="bean" match="org.bibeault.*"/>          <-- (1) Specifies bean classes

    <create creator="new" javascript="CustomerFactory">       <-- (2) Defines JavaScript class
      <param name="class"
             value="org.bibeault.aip.dwr.CustomerFactory"/>
    </create>
  </allow>
</dwr>
```

The DWR documentation should be consulted for all the possible settings that can be made in this file, but essentially we tell DWR that we want it to convert bean classes (such as the one we will return from our `findByName()` method) from our

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

package ❶. Then we define the class that we wish to remote, along with the means of construction (`creator="new"`) and the name of the JavaScript object that will serve as its client-side stub ❷.

Now comes the fun!

Start the web application and "hit" the DWR servlet with no path info. Assuming that the web application's context path is `/aip.chap4`, the URL could be http://localhost:8080/aip.chap4/dwrserver/. Because we enabled the debug mode via the `init` parameter to the servlet, DWR displays a test page that dynamically shows us some very useful information about our DWR environment, as shown in figure 4.6.

This page shows us all the classes that DWR has mapped for us. Clicking the `CustomerFactory` link reveals some useful information about that class, as shown in figure 4.7.

Not only does this page show us the `<script>` elements that need to be included in order to use the class, it also shows us the methods that are declared and even allows us to test them with sample data. The utility of this feature during development cannot be stressed enough!

If we inspect the URLs for the `<script>` elements, we see that even though they look like JavaScript file references, they are actually invocations of the DWR servlet that we declared in the web.xml. These `.js` files do not actually exist anywhere in the filesystem, but are dynamically served from the DWR servlet upon reference. This makes DWR an easy toolkit to use as, except for the servlet and the `dwr.xml` file, there's not much else to keep track of.

If we click the link for the `CustomerFactory.js` file, we see the display in figure 4.8, where we see the local stub created for the methods in our mapped class.

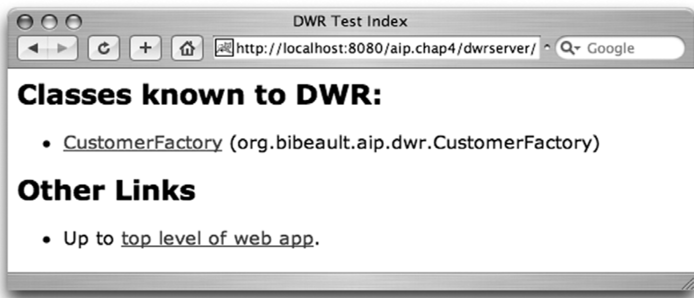

Figure 4.6 DWR test display: list of known classes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

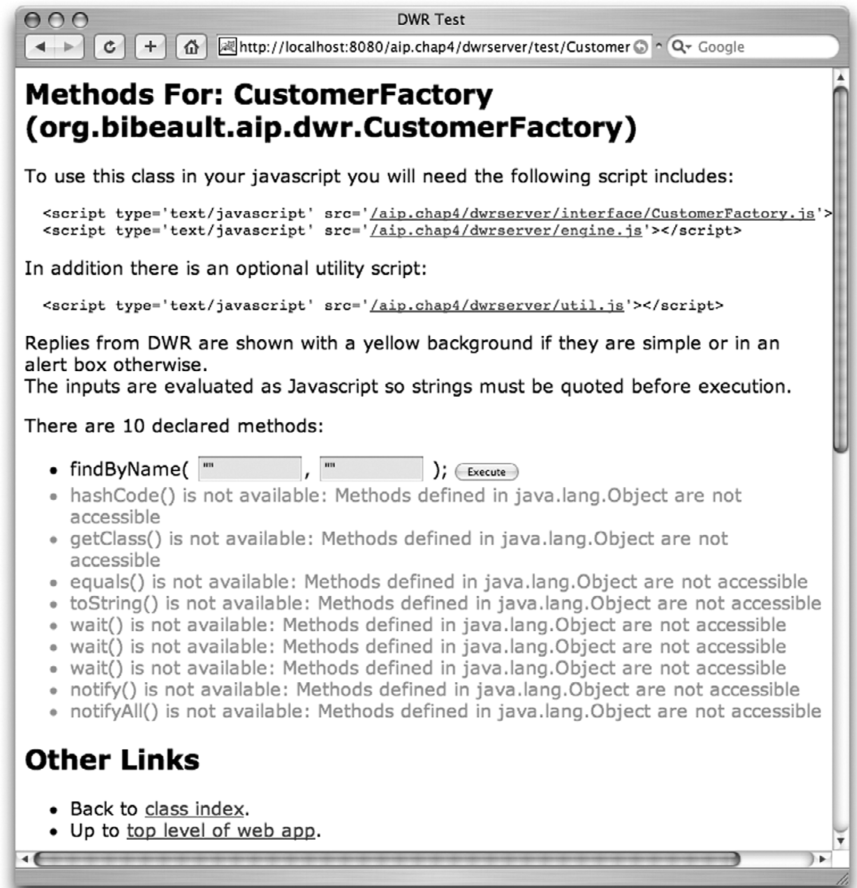

Figure 4.7 DWR test display: class information

So now we're ready to actually code our page. Cutting and pasting the script elements from the DWR-generated page of figure 4.7, we start by adding them to the <head> element of our page:

```
<script type='text/javascript'
  src='/aip.chap4/dwrserver/interface/CustomerFactory.js'></script>
<script type='text/javascript'
   src='/aip.chap4/dwrserver/engine.js'></script>
```

The form to capture the customer data is coded as

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

http://localhost:8080/aip.chap4/dwrserver/interface/CustomerFactory.js

```
function CustomerFactory() { }
CustomerFactory._path = '/aip.chap4/dwrserver';

CustomerFactory.findByName = function(p0, p1, callback) {
    DWREngine._execute(CustomerFactory._path, 'CustomerFactory', 'findByName', p0, p1, callback);
}
```

Figure 4.8 DWR dynamically generated JS stub

```
<form name="customerForm" action="/doSomething">
  <div>
    <label>First name:</label>
    <input type="text" name="firstName"
      onblur="lookupByName(this.form);"/>
    <label>Last name:</label>
    <input type="text" name="lastName"
      onblur="lookupByName(this.form);"/>
  </div>
  <div>
    <label>Address:</label>
    <input type="text" name="address"/>
  </div>
  <div>
    <label>City:</label>
    <input type="text" name="city"/>
    <label>State/Province:</label>
    <input type="text" name="state"/>
  </div>
  <div>
    <label>Postal Code:</label>
    <input type="text" name="postalCode"/>
    <label>Country:</label>
    <input type="text" name="country"/>
  </div>
  <div>
    <input type="submit" value="OK"/>
  </div>
</form>
```

This form is fairly straightforward (if incredibly ugly without any styling) except for the calls to the `lookupByName()` function set up as the `onblur` event handlers for the `firstName` and `lastName` fields:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function lookupByName(form) {                    ← ❶ Passes containing form
  if ((form.firstName.value != '') &&
      (form.lastName.value != '')) {
    CustomerFactory.findByName(form.firstName.value,    ← ❷ Invokes stub for
                               form.lastName.value,         findByName()
                               onCustomerFound);
  }
}
```

In this handler, the containing form is passed ❶ and checks the `firstName` and `lastName` fields to see if they are both populated. If so, the stub for the remote `findByName()` method is invoked ❷.

Note that this stub is similar, but not identical, to the remote method being stubbed. First of all, there is no return value from this function because it is not invoked synchronously. Second, an extra parameter has been added to the method signature to specify the callback function that is to be invoked when the asynchronous call completes. This function will be passed the remote method's return value as its single parameter.

The code for this callback handler is as follows:

```
function onCustomerFound(customer) {
  if (customer != null) {
    var form = document.customerForm;
    form.address.value = customer.address;
    form.city.value = customer.city;
    form.state.value = customer.state;
    form.postalCode.value = customer.postalCode;
    form.country.value = customer.country;
  }
}
```

This callback is invoked with the return value from the remote method. We simply check to make sure that it is not `null` (recall that the remote method returns `null` if a customer cannot be uniquely identified). If it's not `null`, we fill in the form with values from the passed object.

DWR has marshaled the data from the Java Customer class and created a JavaScript object with properties that correspond to each of the JavaBean properties that we defined on our Java Customer class. Displaying the example page in our browser and entering the name of our lone customer results in the display shown in figure 4.9. The entire page, when completed, is shown in listing 4.8.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

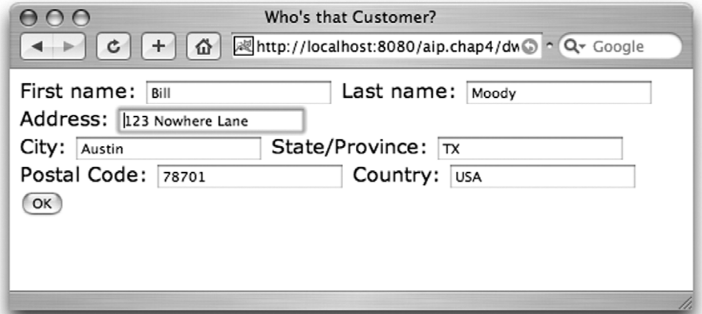

Figure 4.9 Paging Bill Moody!

Listing 4.8 Direct remoting with DWR

```
<html>
  <head>
    <title>Who's that Customer?</title>
    <script type='text/javascript'
      src='/aip.chap4/dwrserver/interface/CustomerFactory.js'>
    </script>
    <script type='text/javascript'
      src='/aip.chap4/dwrserver/engine.js'></script>
    <script>
      function lookupByName(form) {
        if ((form.firstName.value != '') &&
             (form.lastName.value != '')) {
          CustomerFactory.findByName(form.firstName.value,
                                     form.lastName.value,
                                     onCustomerFound);
        }
      }

      function onCustomerFound( customer ) {
        if (customer != null) {
          var form = document.customerForm;
          form.address.value = customer.address;
          form.city.value = customer.city;
          form.state.value = customer.state;
          form.postalCode.value = customer.postalCode;
          form.country.value = customer.country;
        }
      }

    </script>
  </head>

  <body>
    <form name="customerForm" action="/doSomething">
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <div>
          <label>First name:</label>
          <input type="text" name="firstName"
                 onblur="lookupByName(this.form);"/>
          <label>Last name:</label>
          <input type="text" name="lastName"
                 onblur="lookupByName(this.form);"/>
        </div>
        <div>
          <label>Address:</label>
          <input type="text" name="address"/>
        </div>
        <div>
          <label>City:</label>
          <input type="text" name="city"/>
          <label>State/Province:</label>
          <input type="text" name="state"/>
        </div>
        <div>
          <label>Postal Code:</label>
          <input type="text" name="postalCode"/>
          <label>Country:</label>
          <input type="text" name="country"/>
        </div>
        <div>
          <input type="submit" value="OK"/>
        </div>
      </form>
    </body>

  </html>
```

Discussion

This section exposed us to a small fraction of the capabilities of the DWR toolkit. Unlike the other Ajax-capable toolkits we have looked at, DWR abstracts the request-response cycle away in favor of providing an RPC-like means to call Java functions on the server.

While our example is simple, it's easy to envision how such capabilities can be used for a variety of client-side tasks, not limited to mere data lookups. Imagine, for example, the value of these capabilities when performing field data validations that require server-side participation.

Obviously DWR is only useful for Java web applications hosted by a servlet engine. Developers or page designers who are using alternative server-side mechanisms would be best served by using a toolkit that adheres to the typical request-response cycle.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

4.5 Summary

In this chapter we introduced a number of open source libraries that make it easier to use Ajax in our web applications, or that use Ajax to provide asynchronous communication between a client browser and the server.

Some of these libraries, such as the Dojo toolkit and Prototype, provide a thin wrapper around Ajax calls that make them easier to code and maintain. Others, such as jQuery, provide a substantial number of methods that make it easy to perform some of the most common Ajax interactions. Still others, such as DWR, provide a different paradigm from the typical HTTP request-response cycle using Ajax as the transport mechanisms. These libraries are just a few of the many freely available on the Internet.

Time, space, and other practical considerations prevent us from examining more of the available libraries. If this sampling has piqued your interest, here are some others that you may wish to investigate:

- Scriptaculous (http://script.aculo.us/), another JavaScript library based on Prototype, very closely aligned with the Ruby on Rails project
- Rico (http://openrico.org/), a JavaScript library for creating rich Internet applications that includes support for drag and drop, Ajax, and cinematic effects
- Echo2 (www.nextapp.com/), a platform for developing web applications intended to provide the rich capabilities of desktop clients
- Sarissa (http://sarissa.sourceforge.net/), an XML and XSLT-centric library that acts as a cross-browser wrapper for native XML APIs
- Sajax (www.modernmethod.com/sajax/), a JavaScript Ajax framework intended for web applications developed with PHP, Perl, or Python
- ThinkCAP JX (www.clearnova.com/), a RAD development tool for Ajax-enabled business applications

And more... This lists only scratches the surface.

So get on out there and look around. This is an exciting and rapidly growing area. By the time this book reaches print, even more new and exciting toolkits may have hit the Web!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Part 2

Ajax Best Practices

Part 2 presents nine chapters, each of which dives into an area of web application development essential for Ajax programs.

Chapter 5 begins part 2 with an in-depth look at event handling in the browsers. We discuss models of event handling and explain how to establish event handlers for the various event types. We identify browser issues and tactics to ease cross-browser coding.

In chapter 6, we develop ways to validate form-data entry values using the event-handling lessons of the previous chapter. We examine a validation framework, and you'll learn how to hijack form submissions to avoid full-page refreshes.

Navigating application content is the subject of chapter 7. Menus, trees, accordion controls, tabs and toolbars are all discussed. We include peeks at OpenRico and qooxdoo in the example code for this chapter.

The pain of dealing with users who insist on hitting those back and refresh browser controls is addressed in chapter 8. We describe tactics for hiding these controls, as well as strategies for dealing with them when they're not hidden.

Chapter 9 focuses on adding drag-and-drop capabilities to web applications. We enlist the aid of the Scriptaculous library to sort lists using drag and drop, and then examine a simple drag-and-drop shopping cart. A look at ICEfaces rounds out the chapter.

Usability concerns, particularly those commonly associated with Ajax applications, are explored in chapter 10. We develop strategies for dealing with latency issues and discuss how to alleviate user frustration.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Chapter 11 takes a look at maintaining client state, caching and prefetching data, and other topics in the realm of state management.

Rousing web service open APIs such as Yahoo! Maps, Yahoo! Geocoding, Yahoo! Traffic, Google search, and Flickr photo services are investigated in chapter 12. We devise a means to circumvent the dreaded cross-browser security limitations, and you'll learn how to make RESTful requests to these exciting services via Ajax.

Finally, chapter 13 ties everything together to create a fully functional "mashup" web application employing the Yahoo! and Flickr open APIs.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

5 Handling events

This chapter covers

- Models of browser event handling
- Commonly handled event types
- Making event handling easier
- Event handling in practical applications

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The days of boring HTML applications are over now that Ajax allows us to build highly interactive web applications that respond fluidly to user actions. Such user actions may include clicking a button, typing in a text box, or simply moving the mouse. User actions have been translated into events throughout the history of graphical user interfaces (GUIs), and it is no different in the browser world. When a user interacts with a web page, events are fired within the DOM hierarchy that is being interacted with, and if there are event handlers associated with the events fired on the document's elements, they will be called when the events occur. Ajax applications depend heavily on these events and their handlers; they could even be considered the lifeline of every Ajax application.

Before we get ahead of ourselves, let's see how we can add a simple event handler to a web page. In the following code snippet, notice how the `<img>` element has an `onclick` attribute. This attribute defines an event handler that will be called by the browser when the user clicks the mouse on the `<img>` element.

```
<html>
  <body>
    <img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
  </body>
</html>
```

If you load this example into a browser, you will see that when the mouse button is clicked while hovering over the image, the alert box showing the message "Woof!" is displayed, as shown in figure 5.1.

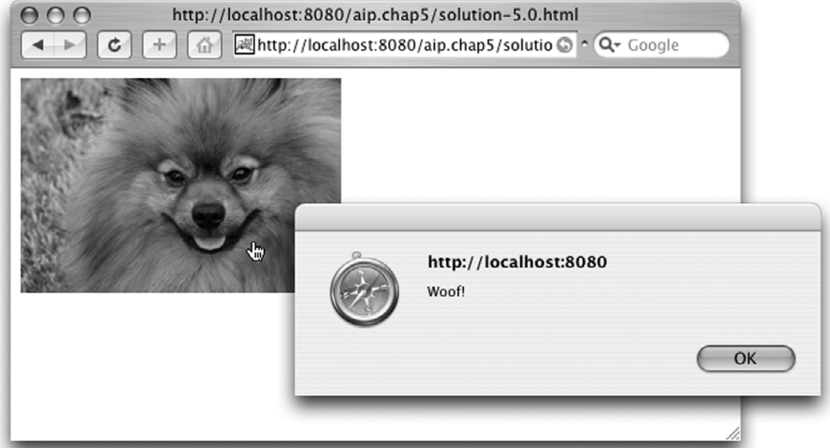

Figure 5.1 Making the dog bark

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This demonstrates how easy it is to assign an event handler to a DOM element. Throughout this chapter we'll examine the major aspects of event handling. We begin by reviewing the various ways in which we can define event handlers using the various models available. We'll see how the process differs across the browser platforms and look at ways to make it portable across browser implementations. We'll also learn about the information about the event made available to event handlers when they are invoked. We'll discuss the concepts of *event bubbling* and *event capturing* that specify how events are propagated through the DOM, and we'll also look closely at the commonly handled event types. Finally, we'll whip up some real-world examples that demonstrate how we can put these concepts to use in our applications.

5.1 Event-handling models

While we've seen how easy it is to *declare* simple event handlers, you would think that writing event handlers should be just as easy. We just write some script into the handler attributes and the browser executes it when the event occurs. What could be simpler? But we wouldn't need this chapter if it were really that simple, would we?

In the present-day world, there are three event models that we need to contend with in order to use events in our web applications:

- The Basic Event Model, also informally known as the DOM Level 0 Event Model, which is fairly easy, straightforward, and reasonably cross-platform.
- The DOM Level 2 Event Model, which provides more flexibility but is supported only on standards-compliant browsers such as Firefox, Mozilla, and Safari.
- The Internet Explorer Event Model, which is functionally similar to the DOM Level 2 Model, but which is proprietary to Internet Explorer.

First we'll take a look at registering and writing handlers using the basic model, and then we'll look at using the two advanced models.

5.1.1 Basic event-handling registration

The example we examined in the chapter introduction illustrates the use of the Basic, or DOM Level 0, Model. This is the oldest approach to event handling and enjoys strong (though not complete) platform independence. It is well suited for basic event-handling needs. And as we'll see, it's not completely replaced by the more advanced models, but is typically used in conjunction with those models.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This model allows event handlers to be assigned in one of two ways:

- Inline with the HTML element markup, using event attributes of the HTML elements
- Under script control, using properties of the DOM elements

Recall the `<img>` element from our small example:

```
<img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
```

This is an example of using the *inline* technique.

The value of the `onclick` event attribute becomes the body of an anonymous function that serves as the handler for the click event. While this is easy, it has its limitations.

The best-practice design approach to building web applications separates the view of the application (HTML) from its behavior (JavaScript). Using the inline approach of defining event handlers violates this principle, and therefore it is generally recommended that use of inline handler declarations be limited or avoided.

The better approach is to attach the event handler to the DOM element under script control. This technique has become more prevalent in recent years, as the browser DOM has become more standardized and JavaScript developers have become more familiar with it. All DOM elements have properties that represent the events that can be fired on the element: for example, `onclick`, `onkeyup`, or `onchange`.

Let's rework the sample code that we saw earlier into a complete HTML document and programmatically set the `onclick` event handler of the image as shown in listing 5.1.

Listing 5.1 Assigning an event handler in script

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
      };
    </script>
  </head>
  <body>
```

(1) Declares the page's onload handler

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    <img src="image.jpg" id="anImage"/>     <-- 2 Declares the script-free image element
  </body>
</html>
```

If you have downloaded the source code that accompanies this chapter from www.manning.com/crane2, you'll find this HTML document in the file `chap5/listing-5.1.html`.

While this example is functionally equivalent to our previous example, it exhibits a higher level of sophistication than the previous code. We've separated the behavior from the view by factoring the script out of the `<body>` element ❷ and into a `<script>` element in the `<head>`. Note that we have placed the code in yet another event-handler function: the `onload` event handler ❶ for the page.

Although this seems like more code to do the same thing that we saw in the first example, this technique not only improves the structure of the page but also gives us more flexibility.

An important aspect of that flexibility is the ability to control *when* handlers are established and removed. With the inline method, we're limited to establishing handlers when the page loads, and those handlers exist for the duration of the page. Assigning the handler under script control allows us to establish a handler whenever we want to. In the example of listing 5.1, we chose to establish the handler when the page loads, but we could just as easily have deferred that action until a later time as the result of some other event. Moreover, we can *remove* the event handler at any time by assigning `null` to the event property—something we can't do with inline handlers.

In our example, we created the event handler using an anonymous function literal—after all, why create a separate named function if we don't have to? But when assigning named functions as event handlers, it is important to remember not to include parentheses after the function name. We want to assign a *reference* to the function as the property value, not the result of *invoking* the function! For example, the following will invoke a function named `sayWoof()` rather than setting it as the event handler. Don't make this common mistake.

```
element.onclick = sayWoof(); //Wrong!

element.onclick = sayWoof; //Correct!
```

Although the DOM Level 0 Event Model is somewhat flexible, it does suffer from limitations; for example, it doesn't easily allow chaining of multiple JavaScript functions in response to an event.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

So how would we register two functions to handle a single event? Let's initially take a rather naive approach and modify our example by adding two JavaScript event handlers to the `onclick` property of the `<img>` element, as shown in listing 5.2 (found in the file `chap5/listing-5.2.html` in the downloadable source code) with the added code highlighted in bold.

Listing 5.2 Attempting to assign two handlers

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
        document.getElementById('anImage').onclick = function() {
          alert('Woof again!');
        }
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

When we run this code, it is obvious that only the second handler is called because only a single alert containing "Woof again!" is displayed. Looking at the code, this shouldn't be much of a surprise. Since `onclick` is simply a property of the `<img>` element, multiple assignments to it will overwrite any previous assignment, just as with any other property.

This poses an interesting question: is it possible to call multiple functions in response to an event? Using the DOM Level 0 Event Model, there is no means to register multiple event handlers on the same event by assigning the handlers to the element's event properties. We could factor the code from multiple functions into a single function, or we could write a function that in turn called the other functions. But each of these tactics is a rather pedestrian approach and is not very scalable. If we had no other recourse, a more sophisticated means to accomplish this would be to utilize the Observer pattern (also known as the Publisher/Subscriber pattern) in which our registered handler would serve as the observer, and other functions could register themselves as subscribers.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Luckily, we won't have to resort to such shenanigans as the browsers allow us to register multiple handlers—though, unfortunately, not in a browser-independent fashion—if we use the advanced event-handling models. Let's take a look at how to do just that.

5.1.2 Advanced event handling

In a perfect world, code written for one browser would work flawlessly in all other browsers. We don't live in that world. So when it comes to the advanced event models, we need to deal with browser differences. On the one hand, there is the World Wide Web Consortium (W3C) way of doing things, and then there is the Microsoft way of doing things. Let's look at the standardized W3C way first.

For browsers that adhere to the DOM Level 2 Event Model, a method named `addEventListener()` is defined for each DOM element and can be invoked to add an event handler to that element. This method accepts three arguments: a string declaring the event type, the event-handler function to be executed (also known as the *listener*), and a Boolean value denoting whether or not event capturing is to be enabled. We'll explain this last argument when we discuss event propagation, but for the time being, we'll just leave it set to `false`.

The event type argument expects a string containing the name of the event type to be observed. This is the attribute name for the event with the `on` prefix omitted—for example, `click` or `mouseover`.

Let's change our sample code of listing 5.2 to use this method. We'll replace the basic means (which sets the `onclick` property of the element) with calls to the `addEventListener()` method, as shown in listing 5.3 (with changes highlighted in bold).

Listing 5.3 Adding an event handler the W3C way

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof!'); },
          false);
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof again!'); },
          false);
      };
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

When *this* page is displayed and the image is clicked, both the alert boxes show up without resorting to hokey container functions to chain both event handlers. Note that this code does not work in Internet Explorer; later in this section we'll see how IE implements advanced event handling in its proprietary fashion.

Also note that, when multiple handlers for the same event on the same elements are established as we have done in our example, the DOM Level 2 Event Model does not guarantee the order in which the handlers will be executed. In testing, it was observed that the handlers seemed to be called in the order that they were established, but there is no guarantee that will always be the case and it would be folly to write code that relies on that order.

To remove an event handler from an element, we can use the `removeEventListener()` method defined for the DOM elements.

The proprietary Microsoft means of attaching events is similar in concept, but different in implementation. It uses a method named `attachEvent()` defined for the DOM elements to establish event handlers. This function accepts two arguments: the event name and the event-handler function to be executed. Unlike the event type that is used with `addEventListener()`, the event property name, complete with the `on` prefix, is expected.

Armed with this information, let's modify our sample code once again. We'll add some detection to our code and use the method that's appropriate to the containing browser. The updated code is shown in listing 5.4 (available in the downloadable source code for this chapter), once again with changes highlighted in bold.

Listing 5.4 Doing it either way

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        if (document.getElementById('anImage').attachEvent) {
          document.getElementById('anImage').attachEvent(
            'onclick',
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          function() { alert('Woof!'); });
        document.getElementById('anImage').attachEvent(
          'onclick',
          function() { alert('Woof again!'); });
      }
      else {
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof!'); },
          false);
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof again!'); },
          false);
      }
    }
  </script>
</head>
<body>
  <img src="image.jpg" id="anImage"/>
</body>
</html>
```

In the first line of the `onload` event handler, we check to see which method we should use. Note the use of a test known as *object detection*. Rather than testing for a specific browser, we check to see if the proprietary `attachEvent()` method exists on the element. If so, we use it; otherwise, we use the standardized W3C method.

When we display this page in any browser, it is guaranteed to work as long as the browser supports either one of these mechanisms. When we click on the image when displayed in Internet Explorer, we notice something strange: the alerts are shown in the reverse order! Or maybe not. Truth be told, as with the DOM Level 2 Event Model, we don't know in which order they will be shown. The definition of the `attachEvent()` method clearly states that multiple event handlers attached to the same event type on an element will be triggered in random order.

This completes our exploration into the ways in which event handlers can be registered across the different browsers. You saw the ease with which we can use the inline technique as well as its disadvantages. The DOM Level 0 means of registering event handlers is portable across browsers, but does not provide an automatic way of chaining multiple event-handler functions. We showed you how to attach event handlers in a more advanced way using either the DOM Level 2 or Internet Explorer models. Although this approach is flexible and allows us to dynamically attach, detach, and chain event handlers, it suffers from cross-browser issues,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

forcing us to resort to object detection in order to call the method appropriate to the current browser. Fortunately, frameworks are available that abstract all these differences away and help us write code that is portable across all supported browsers. We'll see how using Prototype helps us in this manner in section 5.3.

Before we do that, let's build on our foundations of event handling in general. In the next couple of sections you'll see in detail how event information is made available to an event handler and how events are propagated through the DOM tree.

5.2 The Event object and event propagation

Two other important topics that we need to understand when dealing with events in the browser are the Event object and the manner in which events are propagated. The Event object, actually an instance of the Event class, is important for obtaining information about the event, and event propagation defines the order in which an event is delivered to its observers. First let's tackle the Event object.

5.2.1 The Event object

When an event is triggered, an instance of the Event class is created that contains a number of interesting properties describing that event. In our event handlers, we typically want to access that Event object to obtain interesting properties such as the HTML element on which the event occurred, or which mouse button was clicked (for mouse events). As with much else in the world of events, this Event object instance is made available to the event handlers in a browser-specific fashion.

For standards-compliant browsers, the Event object instance is passed as the first parameter to the event-handler function. In Internet Explorer, the instance is attached as a property to the window object instance (essentially a global variable).

Let's explore what it takes to deal with this object. Since we're getting tired of the alerts, let's also change the code to write diagnostic information into a `<div>` element below the image, as shown in listing 5.5.

Listing 5.5 Grabbing the Event instance

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick =
          function(event) {
            if (!event) event =    <-1 Grabs event object instance
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
                window.event;            ❷ Obtains event target element reference
              var target =            <┘
                event.target ? event.target : event.srcElement;
              document.getElementById('info').innerHTML +=
                'I woof at ' + target.id + '!<br/>';
            }
          }
        </script>
      </head>
      <body>
        <img src="image.jpg" id="anImage"/>
        <div id="info"></div>
      </body>
    </html>
```

In this example, we obtain a reference to the instance of Event by checking first to see if the parameter passed to the event-handler function, which we cleverly named `event`, is defined (as it will be for standards-compliant browsers) and if not, copies the `event` property from the window object ❶ where IE will have placed it.

We then want to obtain a reference to the *target element* ❷—that is, the element for which the event was generated. Again, we need to do so in a browser-specific manner as the definition of the Event class differs between IE and standard browsers.

We check to see if the standard `target` property is defined, and if not, we use the proprietary `srcElement` property.

What a pain! It seems that almost each and every step of event handling needs to do things differently in order to work in both IE and the browsers that support the W3C standards!

Well, yes, that's pretty much the case. But fear not; help is at hand. But first, let's find out what event propagation is all about.

5.2.2 Event propagation

We've focused, up to this point, on handlers that are directly defined on the elements that trigger the events, as if they are the only handlers that are significant. As it turns out, this is not the case. Rather, the event is delivered not only to the target element, but potentially to all its ancestors in the DOM tree as well. In this section, we'll see how events are propagated through the DOM tree, and learn how we can affect which event handlers are called along the way—and even how to control the propagation of an event.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We'll start by talking about how events are propagated in browsers that follow the DOM Level 2 Event Model. We'll then examine how Internet Explorer supports only a subset of that model.

In standards-compliant browsers that support the DOM Level 2 Model, when an event is triggered, that event is handled in three phases. These phases, in order, are called *capture*, *target*, and *bubble* phases.

During the capture phase, the event traverses the DOM tree from the document root element down to the target element. Any event handlers established on the traversed elements for the type of event that is being propagated are invoked *if* the event handler was registered as a *capture handler*. Remember that third parameter to the `addEventListener()` method that we've been ignoring up until now? If that parameter is set to `true`, the event handler is registered as a *capture handler*. If it's set to `false`, as we have been doing up to now, the event handler is established as a *bubble handler*. Each event handler can be either a capture or a bubble handler, but never both.

Once the event has traversed downward to the target element, activating any appropriate capture handlers along the way, the propagation enters the target phase. During this phase, the event handlers established on the target element itself are triggered as appropriate. If both a capture and a bubble handler are established on the target element, they are both invoked during this phase.

The event propagation then reverses direction and "bubbles" up the DOM tree from the target element to the root element. This is the bubble phase, and along the way, any bubble handlers established for the event type on the traversed elements are triggered.

Enough talk—how about a diagram? Let's say that we modify the body of our example program to nest the `<img>` element within two `<div>` elements as follows:

```
<div id="level1">
  <div id="level2">
    <img src="image.jpg" id="anImage"/>
  </div>
</div>
```

When we click on the image element, the click event is propagated through the DOM tree as shown in figure 5.2.

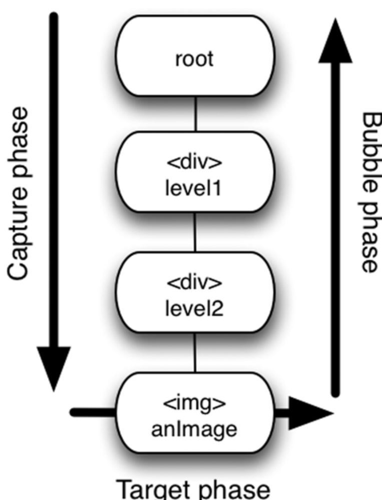

Figure 5.2 Down and up the DOM tree

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Now let's see it in action. Consider the code in listing 5.6.

Listing 5.6 Establishing capture and bubble handlers

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').addEventListener(
          'click', react, false);
        document.getElementById('level1').addEventListener(
          'click', react, true);
        document.getElementById('level2').addEventListener(
          'click', react, false);
      }

      function react(event) {
        document.getElementById('info').innerHTML +=
          'I woof at ' + event.currentTarget.id + '!<br/>';
      }
   </script>
  </head>
  <body>
    <div id="level1">
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

❶ Establishes handlers
❷ Defines handler function
❸ Defines nested element

In this example, we've modified the body ❸ as described earlier, nesting the `<img>` element within two `<div>` elements.

Within the `onload` event handler ❶, we establish three event handlers: one on the `<img>` element, and one on each of the nesting `<div>` elements. Note that the event handler established on the element with the `id` of `level1` is registered as a capture handler by way of its third parameter.

All event handlers are assigned the same function, `react()` ❷, which emits a message that contains the value of the `currentTarget` property of the passed `event` instance. This property differs from the `target` property in that the `target` property identifies the element that triggered the event while `currentTarget` identifies the element that is the current subject of the event propagation—in other words, the element upon which the handler was established.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Before looking at figure 5.3, try to guess what the order of handler invocation will be. Did you get it right?

When we display this example in a standards-compliant browser (remember, the code we're using is not suited for Internet Explorer yet) and click the image, we see the display shown in figure 5.3.

The reason for the order of the output should be clear. The handler established on the `level1` element is a capture handler, while the rest are bubble handlers. The `level1` handler triggers, emitting its output, during the capture phase; the event handler on the `<img>` element triggers during the target phase; and finally, the event handler on `level2` is invoked during the bubble phase.

Internet Explorer supports only the target and bubble phases; no capture phase is supported. To modify this example for IE, we need to change the calls to the `addEventListener()` method to `attachEvent()` and alter the event-handler function as well. Unfortunately, there is no property corresponding to `currentTarget` in the Event class provided by Internet Explorer.

If you are targeting IE, and getting a reference to the current target element of the bubble phase is essential to your requirements, you'll need to come up with some underhanded means of getting a reference to that element to the event handler. One tactic that we could employ would be to use the Prototype `bind()`

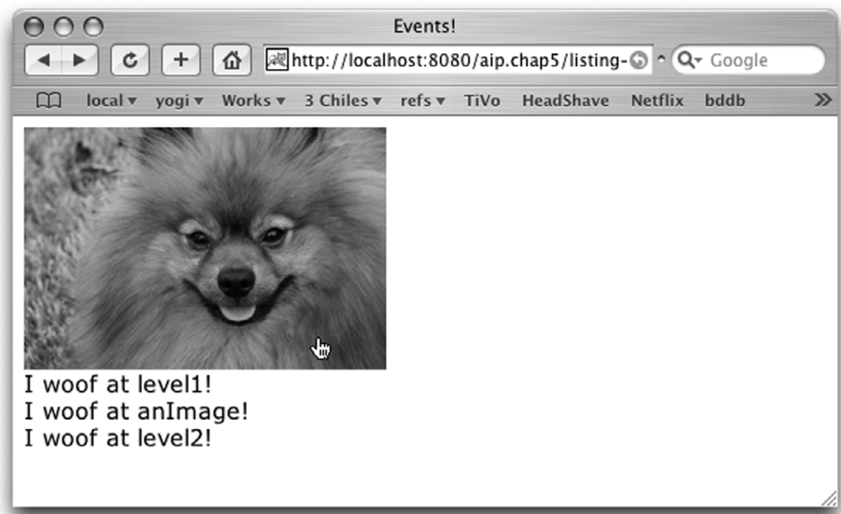

Figure 5.3 Result of capture and bubble

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

mechanism to force the function context object (the `this` reference) for the event handler to be the element upon which the handler is being established, as in

```
Event.observe('someId','click',someHandler.bind($('someId')));
```

Then, in the event handler, we could add

```
if (!event.currentTarget) event.currentTarget = this;
```

This would detect environments where `currentTarget` is not defined and set the context object reference into the Event instance to be used in a browser-independent fashion in the remainder of the handler. A bit Byzantine, perhaps, but useful if you absolutely must have this information available across all browsers.

Stopping propagation

There are times when you may want to prevent an event from continuing its propagation. An example is when you know that you have handled the event as much as you require and allowing the event to further propagate would trigger unwanted handlers.

In a standards-compliant browser, the `stopPropagation()` method of the Event class would be called within an event handler to prevent further propagation of the current event. In IE, the `cancelBubble` property of the Event instance is set to `true`. It may seem odd to set a property, rather than call a method, in order to effect a stop to the propagation, but that's how IE defines this action.

Preventing the default action

Some events, known as *semantic events*, trigger a default action in the browser—such as when a form is submitted, or when an anchor element is clicked.

In DOM Level 0 handlers, the value `false` can be returned in order to cause that default action to be canceled. In DOM Level 2 handlers, the `preventDefault()` method of the Event class serves the same purpose. Calling this method prevents the default action from taking place. This can be used, for example, to prevent a form from being submitted if a validation check conducted by a submit event handler determines that one or more form fields are not valid. In IE, the `returnValue` property of the Event instance is set to `false` to prevent the browser from carrying out the default action.

All these browser differences are a royal pain to deal with. Luckily, we're not the only ones who think so, and those who write JavaScript libraries have come to our aid. Let's take a look at how a now-familiar library makes event handling less painful in our pages.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

5.3 Using Prototype for event handling

Several JavaScript libraries are available that simplify the process of defining event handlers by abstracting browser differences away. Prototype, which we examined previously in chapters 3 and 4 with regard to helping us write object-oriented JavaScript and make Ajax requests, also provides a simple but convenient abstraction to help us with event handling.

Prototype defines an `Event` namespace that possesses a handful of useful methods; the two most important ones are `observe()` and `stopObserving()`. The `observe()` method allows you to attach an event handler to an element, while `stopObserving()` removes event handlers from those elements.

Let's take our example of listing 5.6 and modify it using Prototype. The result is shown in listing 5.7.

Listing 5.7 Event handlers the Prototype way!

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {                      ❶ Defines event handlers
        Event.observe('anImage', 'click', react, false);
        Event.observe('level1', 'click', react, true);
        Event.observe('level2', 'click', react, false);
      }

      function react(event) {                           ❷ Declares handler function
        $('info').innerHTML +=
          'I woof at ' + Event.element(event).id + '!<br/>';
      }
    </script>
  </head>
  <body>
    <div id="level1">
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

What a difference Prototype makes! Not only were we able to use the handy `$()` function that Prototype provides, we were also able to make our example cross-browser compatible while *reducing* the amount of code we had to write.

In the `onload` event handler ❶, we used the `Event.observe()` method to establish our handlers in a cross-browser manner. We are still able to specify, for W3C-compatible browsers, whether the event handler should be a capture or a bubble handler. Under IE, this distinction will just be ignored.

In our event-handler function ❷, we used the `Event.element()` method to obtain a reference to the target element in a browser-agnostic manner.

Note that Prototype does not provide a 100 percent abstraction of the differences between browser event handling. For example, if we wanted to obtain the value of the `currentTarget` property, we'd need to do that directly, and we'd have to be sure to not make such a reference when running within IE. However, Prototype does abstract a great deal of the most commonly used event-handling requirements.

5.3.1 The Prototype Event API

This section provides a quick rundown of the API for the Prototype `Event` namespace, describing each method available.

To begin, the method

```
Event.observe(element,eventType,handler,useCapture)
```

establishes an event handler for the named event type on the passed `element`. The `useCapture` parameter may be omitted and defaults to `false`. This parameter is ignored in IE.

Next, the method

```
Event.stopObserving(element,eventType,handler,useCapture)
```

removes an event handler. The parameters should exactly match those used to establish the handler that is to be removed.

The method

```
Event.unloadCache()
```

removes all handlers established through `observe()` and frees all references in order to make them available for garbage collection. This is especially important for IE, which has a severe memory leak problem with regard to event handling. The best news is that under IE, Prototype automatically calls this method when a page is unloaded.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Next, the method

```
Event.element(event)
```

returns the target element of the passed event.

The method

```
Event.findElement(event,tagName)
```

returns the nearest ancestor of the target element for the passed event that has the passed tag name. For example, you could use this to find the nearest `<div>` parent of the target element by passing the string "div" as the `tagName` parameter.

The method

```
Event.pointerX(event)
```

returns the page-relative horizontal position of a mouse event, and the method

```
Event.pointerY(event)
```

returns the page-relative vertical position of a mouse event.

The method

```
Event.isLeftClick(event)
```

returns `true` if a mouse event was a result of a click of the primary mouse button.

Finally, the method

```
Event.stop(event)
```

stops the event from propagating any further *and* cancels any default action associated with the event.

There! That should make coding for events a lot simpler for us. Now let's turn our attention to the various event types that we commonly need to deal with.

5.4 Event types

When we consider a web application, we know that most events of interest to us occur as the result of the user interacting with the application using the mouse or the keyboard. These events are fired in the DOM element tree in response to user actions such as causing the page to load, clicking a button, moving the mouse, dragging the mouse, typing on the keyboard, or taking an action that would cause the page to unload. As we have seen, we can write event handlers for these events so that our application can respond to these actions. We'll take a closer look at the more commonly handled event types in this section, and we'll start by looking at the mouse events.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

5.4.1 Mouse events

The mouse events that are most commonly handled in a web application are `mouseup`, `mousedown`, `click`, `dblclick`, and `mousemove`. When a user clicks on an element, three events are fired: `mousedown`, `mouseup`, and `click`. Let's observe this firsthand by inspecting the code in listing 5.8.

Listing 5.8 Mouse events on a single click

```
<html>
  <head>
    <title>Mouse events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {                          ❶ Establishes mouse event handlers
        Event.observe('anImage', 'click', react);
        Event.observe('anImage', 'mousedown', react);
        Event.observe('anImage', 'mouseup', react);
      }

      function react(event) {                               ❷ Emits info about event
        $('info').innerHTML +=
          'I bark for ' + event.type +
          ' at (' + Event.pointerX(event) + ','+
          Event.pointerY(event) + ')!<br/>';
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

In this code, we establish event handlers ❶ for the `click`, `mouseup`, and `mousedown` events on the `<img>` element. When the image is clicked on, the event-handler function ❷ examines the `event` instance and emits output containing the event type, as well as the page-relative coordinates of the mouse cursor at the time of the click. In the browser, we'll see the display shown in figure 5.4.

We can see from these results that when the element is clicked on, the `mousedown` event fires first, followed by `mouseup`, and finally, `click`. As an exercise, add `mousemove` or `dblclick` event handlers, and see how those events are delivered in relation to the other event types.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

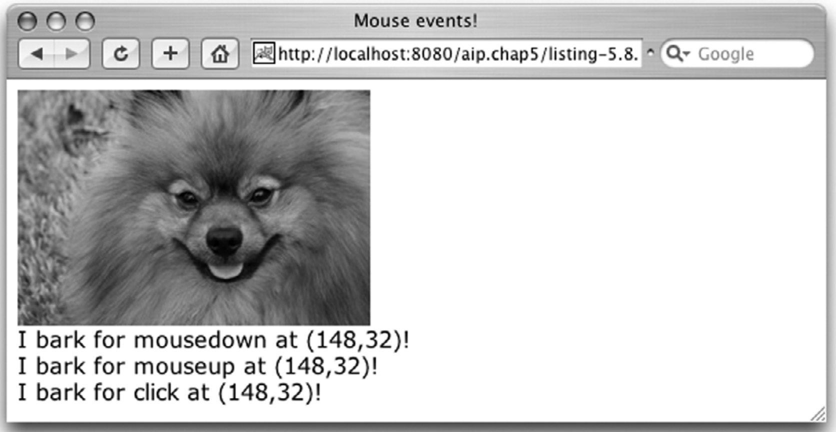

Figure 5.4 Reaction to mouse events

5.4.2 Keyboard events

The commonly handled keyboard events are `keyup`, `keydown`, `blur`, and `focus`. The `keyup` and `keydown` events are similar to the `mouseup` and `mousedown` events; the `keydown` event is fired when the key is pressed, and the `keyup` event is fired when the key is released.

The `focus` and `blur` events are triggered when a DOM element gains or loses focus. In any loaded page, only one DOM element can have focus at a time. The focus can be changed programmatically or as a result of user actions. When a user tabs out of a field, the `blur` event will be fired, followed by the `focus` event of the next element gaining focus. The user can also change focus by clicking on a focusable element.

Let's look at an example of how the `blur` and `focus` events work. Examine the code in listing 5.9.

Listing 5.9 Blur and focus and blur and focus and...

```
<html>
  <head>
    <title>Blur and Focus</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {     <-- (1) Establishes handlers on page load
        Event.observe('nameField', 'blur', react);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      Event.observe('nameField', 'focus', react);
      Event.observe('breedField', 'blur', react);
      Event.observe('breedField', 'focus', react);
      Event.observe('dobField', 'blur', react);
      Event.observe('dobField', 'focus', react);
      $('nameField').focus();    <-- ❷ Assigns focus to first field
    }

    function react(event) {    <-- ❸ Handles blur and focus events
      $('info').innerHTML +=
        Event.element(event).id + ' ' +
        event.type + '<br/>';
    }
  </script>
</head>
<body>
  <form name="infoForm">    <-- ❹ Contains focusable elements
    <div>
      <label>Dog's name:</label>
      <input type="text" id="nameField"/>
    </div>
    <div>
      <label>Breed:</label>
      <input type="text" id="breedField"/>
    </div>
    <div>
      <label>Date of birth:</label>
      <input type="text" id="dobField"/>
    </div>
    <div>
      <input type="submit" id="submitButton"/>
    </div>
  </form>
  <div id="info"></div>
</body>
</html>
```

The structure of this example is similar to the ones that we've been looking at up to this point, but we've made some significant changes in order to shift focus from mouse events (primarily `click`) to keyboard events.

The body of the page has been modified to contain a `<form>` element ❹ in which we have defined three text fields. In the `onload` event handler ❶, we establish a `focus` event handler and a `blur` event handler for each of the text fields. We added these handlers individually for clarity. As an exercise, how would you rewrite this code so that all text fields in a form would be instrumented with the event handlers without having to list them individually?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

At the conclusion of the `onload` handler, we also assign the focus ❷ to the first field in the form under script control. This is significant (besides being a friendly thing to do) because it shows us that when the page loads, the `focus` handler for that first field will trigger. This tells us that the `focus` event is triggered either when focus is assigned by script or when assigned via user activity.

This is not true for all events. The `submit` event for a form element, for example, will not be triggered when a form is submitted under script control.

We've also slightly modified our `react()` ❸ event-handler function to emit the name of the target element followed by the event type.

When this page is initially loaded into the browser, we see the display as shown in the top portion of figure 5.5. As you can see, an invocation of the `focus` event

Figure 5.5 Focusing and blurring

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

handler has already taken place because we assigned focus to the `nameField` element in the `onload` event handler.

After filling in some data and tabbing to the `dobField` element, we can see that as we tab out of each field, the `blur` event handler is called for the element that we are leaving, and the `focus` event handler is triggered as the next element in the tab order gains focus (we'll be seeing a lot more regarding tab order in chapter 10).

Make a copy of the example code in listing 5.9 and add event handlers for the other keyboard events to text fields. Observe how they are triggered as you type the values into the fields.

5.4.3 The change event

We have seen how we can use a `blur` event handler to be notified when the user leaves an element. But it would also be useful to know whether the value of a DOM element has changed when it loses focus—for example, if we want to perform validation on a field only when its data has changed instead of every time it loses focus. For certain types of elements, such as `text`, `textarea`, `select`, and `file`, the DOM fires a `change` event when an element loses focus and the content of the element has changed between the time that field gains and loses focus.

To see this in action, we'll modify our previous example to add change event handlers to the text field elements. The result is shown in listing 5.10, with changes from listing 5.9 highlighted in bold.

Listing 5.10 Knowing what's changed

```
<html>
  <head>
    <title>Ch-ch-changes</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('nameField', 'blur', react);
        Event.observe('nameField', 'focus', react);
        Event.observe('nameField', 'change', react);
        Event.observe('breedField', 'blur', react);
        Event.observe('breedField', 'focus', react);
        Event.observe('breedField', 'change', react);
        Event.observe('dobField', 'blur', react);
        Event.observe('dobField', 'focus', react);
        Event.observe('dobField', 'change', react);
        $('nameField').focus();
      }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      function react(event) {
        $('info').innerHTML +=
          Event.element(event).id + ' ' +
          event.type + '<br/>';
      }
    </script>
  </head>
  <body>
    <form name="infoForm">
      <div>
        <label>Dog's name:</label>
        <input type="text" id="nameField"/>
      </div>
      <div>
        <label>Breed:</label>
        <input type="text" id="breedField"/>
      </div>
      <div>
        <label>Date of birth:</label>
        <input type="text" id="dobField"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

With very little in the way of changes to the HTML document, we've added the ability to be notified when changes are effected on the text fields in our form.

If we were to load this page into our browser, enter some text into the first field, tab to the second, and then tab to the third without entering text into the second field, we'd see something like figure 5.6. As you can see, a `change` event was triggered just prior to the `blur` event for the name field, whose value was changed as a result of user input, but not for the breed field, which was not changed.

5.4.4 Page events

So far we've seen events that are fired when a user interacts with the elements within a *loaded* page, but the browser can also fire events representing page-level activity. These are called *page events*, and they occur when the document is loaded, unloaded, resized, or scrolled. Although these events sound special, we can capture them just as we do with other events by providing event handlers on the `<body>` element of the page or assigning them via the window object.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

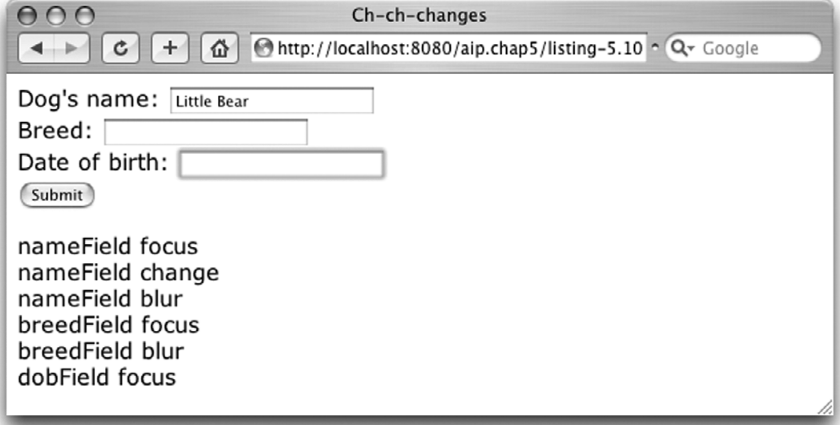

Figure 5.6 What's changed?

In every example we've examined in this chapter, we've already seen the `load` event in action; we used it to declare the other event handlers that we wanted to demonstrate. Now let's add examples of the `unload` and `onbeforeunload` events into the mix, as shown in listing 5.11.

Listing 5.11 Handling page events

```
<html>
  <head>
    <title>Page Events</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {          ❶ Alerts that page is loaded
        alert('Loaded!');
        window.onunload = function() {      ❷ Alerts that page is unloading
          alert('Unloaded!');
        }
        window.onbeforeunload =             ❸ Offers choice
          function() {
            return 'Leaving so soon?';
          }
        }
      }
    </script>
  </head>
  <body>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    <a href="listing-5.11.html">Do it again!</a>
  </body>
</html>
```

As we're going to be loading and unloading the page itself, using on-page output to see what's going on won't work very well, so we've resorted to alert dialog boxes again. In the `onload` event handler, we issue an alert when the page is loaded ❶ and then proceed to establish event handlers for the `unload` and `beforeunload` events.

In the `onunload` event handler ❷, we simply issue another alert that announces that that event has triggered. But the `onbeforeunload` event handler is a bit more interesting.

In the `onunload` event handler, there's not much we can do except react to the fact that the page is unloading, but in the `onbeforeunload` event handler, we can actually affect whether or not the page will unload. If a value is returned, as in our `onbeforeunload` event handler ❸, the browser will display a dialog box that asks the user whether the page should unload. That dialog box contains the value that we returned from the handler as part of its text.

When we load this example into the browser, we get an annoying alert that announces that the page has been loaded. Upon clicking the link on the page, which we've wired to simply display the same page again, we see that the browser triggers our `onbeforeunload` event handler and, as a result of the value we returned from that handler, displays the dialog box shown in figure 5.7.

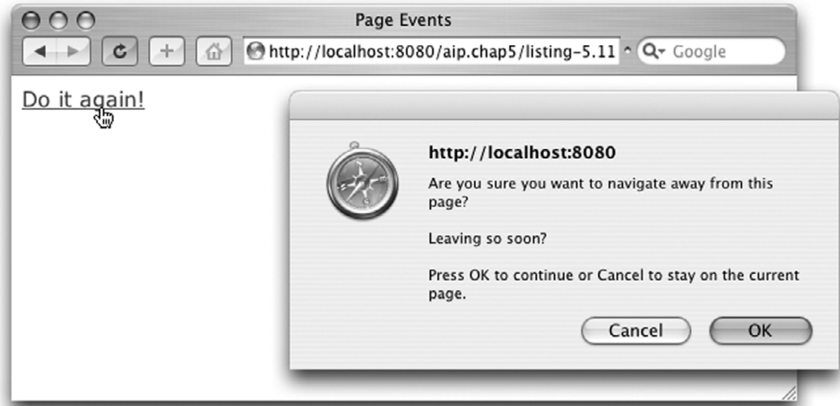

Figure 5.7 Let's chat before you go.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

It doesn't take much imagination to see that this technique could be quite useful for making sure that users don't lose data when they attempt to leave a page before completing their operation. If the user clicks the Cancel button, the page navigation is canceled and the unload operation never takes place. If the user clicks the OK button, the unload operation proceeds and the user receives the alert announcing that the `unload` event handler has been called just before the page reloads.

One aside on the use of the `load` event: it's not uncommon to see pages in which a `<script>` element is placed near the bottom of the page in order to execute code as the page loads. The difference between using this tactic and implementing the `load` event is that the `load` event is guaranteed not to be triggered until after the page has *completed* loading, to include external elements such as script files, CSS style sheets, and images.

That completes our survey of event handling and our examination of some of the most commonly handled event types. Obviously, we haven't explored all events that can be fired within a web page—such an overview could take many chapters—but the information presented here is certainly enough to help you understand how event handling operates and how to handle the event types that are most typically used in modern web applications.

Now that we have a good working knowledge of event handling and the event types, let's take a look at a few practical examples of putting them to work.

5.5 Putting events into practice

The examples in this section require the services of server-side resources in order to execute. To make this as painless and simple as possible for the reader, the sample code for this chapter at www.manning.com/crane2 is already set up to be a complete and runnable web application.

If you are already running a servlet container on your system, simply create a new application context named `aip.chap5` that points to the `chap5` folder of the downloaded code as its document base.

If you are not already running a servlet engine, no need to panic. A PDF document in the `chap4` folder of the download walks you through downloading and configuring Tomcat, and also shows you how to set up application contexts.

When opening these examples in the browser, be sure to address the pages through the web server rather than merely opening the HTML pages as files. For example, to load the example in listing 5.12, you would use the address:

```
http://localhost:8080/aip.chap5/listing-5.12.html
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This assumes, of course, that you are running the servlet container on the default port of 8080. If you've changed that port to another one, be sure to adjust the URL accordingly.

5.5.1 Validating text fields on the server

With the knowledge of how to attach `change` and `blur` event handlers to DOM elements under our belts, it is quite easy to use such handlers to validate input elements on the client to ensure that the data entered is acceptable. Simple client-side checks are easy to conduct, but sometimes business requirements dictate that the data may need to be validated using knowledge that is only available on the server. This may be because the validation is too complex to handle in JavaScript, or because the information that needs to be available in order to validate the data is too vast to send to the page for client-side use.

A common strategy used in classical web applications is to perform the simple validation on the page, and then to perform the more complex validations when the form is submitted. But with the advent of Ajax, we no longer need to put the user through this rather schizophrenic means of validation. To conduct server-assisted validation on the fly, we'll make a server request when a suitable event occurs on the client side, which will validate the data and respond to the client with an appropriate message.

We have all the information we need to solve this problem. We know that we can attach an event to a textbox to detect any changes, and that we can use that event to trigger a request to the server with Ajax. The server-side resource that such a request contacts can validate the data and send back an error message if the data proves invalid.

Note that the purpose of the example in this section is to demonstrate a real-world use of event handling, not to present a mature or sophisticated validation framework. That is a subject that *will* be discussed later in this book in chapter 6 and then again in chapter 10.

Problem

We need to validate text fields using a server-side resource when their value changes.

Solution

We've already seen how to instrument an input text element with event handlers, and this solution will do no differently. The question is: do we trap `blur` or `change` events?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The answer depends on the nature of the data and of the validations to be performed. Since we are going to be making a server round-trip whenever we want to perform a server-assisted validation operation, we want to make sure that we're not firing off requests any more than we need to.

If we know that the data is valid to begin with, we can limit ourselves to trapping `change` events. After all, there's no need to validate data that we know is already good. But in the more common case where fields may start off with unknown data (or even empty), we probably need to trap `blur` events so that the field can be validated every time it is visited.

Establishing an event handler for the field to be validated is as simple as this:

```
Event.observe('fieldId','blur',validationFunction);
```

Listing 5.12 shows a page with a small form consisting of fields for a U.S. address, city, state, and zip code. Our business requirements dictate that the zip code and address must match. This requires consulting a server-side API that the United States Postal Service (USPS) makes available and that must be consulted in the server-side code. Let's see how we handle that on the page.

Listing 5.12 Validating the zip code

```
<html>
  <head>
    <title>I Need Validation</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {          ❶ Sets up event handling
        Event.observe('zipCodeField','blur',validateZipCode);
        $('addressField').focus();
      }

      function validateZipCode(event) {     ❷ Initiates validation request
        new Ajax.Request(
          '/aip.chap5/validateZipCode',
          {
            method: 'get',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              if (transport.responseText.length != 0)
                alert(transport.responseText);
            }
          }
        );
      }
    </script>
  </head>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  <body>
    <form id="infoForm">    <-3 Sets up data entry form
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

Three major activities are addressed by this page: setting up the event handling ❶, reacting to the `blur` event by initiating the validation request to server-side resource ❷, and setting up the data entry form ❸ for the user to fill in.

In the `onload` event handler ❶ for the page, we set up the handler for the `blur` event so that the `validateZipCode()` function will be called whenever the user leaves the zip code field. This function ❷ fires off a Prototype-assisted Ajax request to a server-side resource named `validateZipCode`. As you'll see in a moment, this resource is a Java servlet that does some simplistic hand waving in order to emulate an actual zip code validation operation.

To this resource, we pass the fields of the our form utilizing the handy `serialize()` method that Prototype conveniently adds to the `<form>` element.

The server-side validation resource is defined to return an empty response if all is well and to return an error message if validation fails. So in the `onSuccess` event handler for the Ajax request, we test the text of the response and emit a simple alert if the field failed validation. Remember, more sophisticated validation handling is something that we'll explore in later chapters.

Load this page into a browser (be sure to use the web server URL, not the File menu) and fill in the fields. Note that when you leave the Zip Code field, an alert is issued displaying the validation failure message, as shown in figure 5.8.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

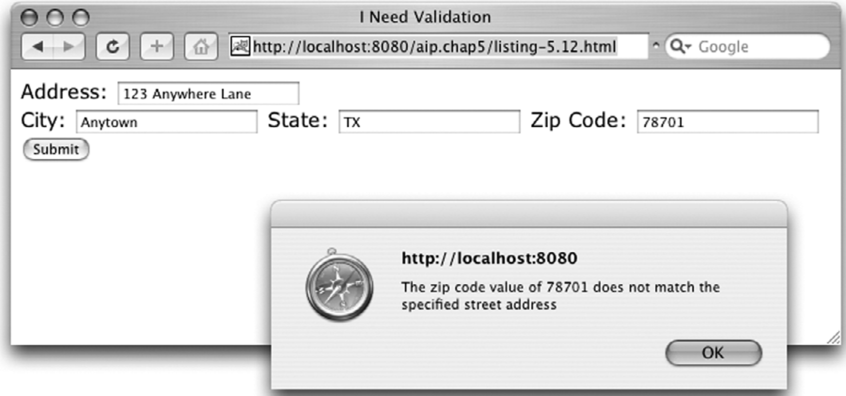

Figure 5.8 Zip code invalid!

In fact, you'll find that every zip code that you type in will generate a validation warning unless you just happened to guess the one valid zip code value of `01826`. That's because our server-side validation servlet is, of course, not really connecting to the USPS database in order to perform an actual validation. The servlet code that is faking a validation operation appears in listing 5.13.

Listing 5.13 Faking our way through a zip code validation

```
package org.aip.chap5;

import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Smoke-and-mirrors validator servlet for listing 5.12. The
 * zip code must be non-blank and equal to "01826" to be
 * considered valid.
 */
public class ZipCodeValidatorServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException {
    StringBuilder result = new StringBuilder ();
    String zipCodeValue = request.getParameter("zipCode");
    if (zipCodeValue.length() == 0) {
      result.append("The zip code field cannot be blank");
    }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      else if (!zipCodeValue.equals("01826")) {
        result
          .append("The zip code value of ")
          .append(zipCodeValue)
          .append(" does not match the specified street address");
      }
      response.getWriter().write(result.toString());
    }
  }
```

There's really not too much to comment on here, except that if this were an actual validation resource, all the fields for the form would be gathered, and a USPS-provided API would be utilized to perform the actual validation. Because that's not the focus of this example (or even of this book), we're just supplying a fake resource that allows us to see our client-side code in practice.

Discussion

In this section, we saw a hybrid method of using client-initiated, server-assisted validation that enables us to give users immediate feedback regarding their entered data, regardless of whether the validation needs server resources.

We used the `blur` event to detect when a user left a field in order to initiate the check. But could we be smarter about this? Once the data has been checked the first time, there's no need to go through the overhead of another server round-trip unless the data has changed. How would you modify the code to only initiate the server check if the validity of the data is unknown?

This hybrid approach of using both client-side and server-assisted on-the-fly validation is a powerful addition to our web application toolbox. Such immediate validation can prevent a lot of user frustration resulting from being told *after* the form submission that there are problems with the submitted data. So by all means, you should implement such validation. But you can never *rely* on it!

Our client-side code is readily available to anyone visiting our pages, and nefarious types will find it easy to reverse-engineer this code to submit their own false data, totally bypassing any client-side validations framework no matter how cleverly crafted. To be sure that the data is valid, *always* implement server-side validation upon form submission regardless of how much validation has been performed prior to that point. You can leverage the same code that you use for client-initiated, server-assisted validation (such as the code we examined in this example) for the final submission-time checks.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Speaking of form submission, there may be times when we want to submit a form to the server without the overhead of a complete page reload. Let's examine that next.

5.5.2 Posting form elements without a page submit

The vast majority of web pages that accept input today are written using the classical technique of submitting a form to the server when data entry is complete. This entails a complete page refresh, which may be undesirable in the context of the rich web applications that we can now deliver using Ajax.

Problem

We want to post a form to a server resource without a full-page reload.

Solution

As it turns out, the solution is almost completely trivial. In fact, we've already pretty much accomplished this task in our previous example. To "submit" the form, we'll use the same technique that we utilized in that example to send form elements to the server for validation.

Trivial and familiar as this solution might be, a few nuances make this problem worth considering. We'll take the code of our previous example, remove the validation check (so that we can focus on the submission topic), and rewire it to hijack the form-submission process in order to send the form to the server under Ajax control rather than as a normal form submission. The results are shown in listing 5.14.

Listing 5.14 Hijacking the submission process

```
<html>
  <head>
    <title>Submit!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">          ❶ Establishes submit event handler
      window.onload = function() {     <-┘
        Event.observe('infoForm','submit',submitMe);
        $('addressField').focus();
      }

      function submitMe(event) {   <-❷ Submits form under Ajax control
        new Ajax.Request(
          '/aip.chap5/handleSubmission',
          {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            method: 'post',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              $('info').innerHTML = transport.responseText;
            }
          }
        );
        Event.stop(event);
      }
    </script>
  </head>

  <body>                                       ❸ Assigns normal submission action
    <form id="infoForm"    <-
          action="/aip.chap5/shouldNotActivate">
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

The changes to this page are subtle but significant. First, we've added a handler to the form for the `submit` event in the window's `onload` handler ❶, which will cause the `submitMe()` function to be called when the form is submitted ❷.

We'll deal with that function in just a minute, but first take a look at the change we made to the `<form>` element ❸. We added an `action` attribute that specifies a server-side resource that does not exist. By doing so, we'll quickly know if our form is ever submitted using the normal default action: the browser will display an unmistakable error page when the server reports that the resource cannot be found.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The `submitMe()` function, called when the `submit` event is triggered, initiates an Ajax request similar to the one we saw in the previous example. But in this case, we specified an HTTP method of `'post'` rather than `'get'`. The heavy lifting is done by the Prototype `serialize()` method.

The server-side resource for the request is a servlet that collects the request parameters and formats a response that contains an HTML snippet showing the names and values of those parameters. (As its operation is not germane to this discussion, we won't inspect it here. But if you're curious, you'll find the source code for the servlet in the downloadable code as the org.aip.chap5. ParameterInspectorServlet class.) This response body is displayed on the page in the `info` element.

Finally, the following statement is executed:

```
Event.stop(event);
```

This Prototype method stops the event from propagating any further and cancels the default action of the event, which in this case is the form submission. Without this statement, the form would go on submitting to the resource identified by the form's `action` attribute.

Discussion

Although this example didn't cover much new ground, it did point out some important concepts, such as using the `submit` event to prevent the submission of the form. We used an event handler and the Prototype event methods for this purpose, but if all you're trying to accomplish is preventing form submission, you can use the following form declaration to return `false` from a DOM Level 0 handler:

```
<form id="my Form" action="whatever" onsubmit="return false;">
```

In our example, we also relied heavily on the services of the Prototype `serialize()` method. This method marshals all the values of the containing form's elements and constructs either a query string or an object hash from those parameters. Because we specified `true` as the parameter to this method, it returns an object hash, which is the preferred technique for Prototype 1.5.

When this page is loaded, data entered, and the Submit button clicked (or the Enter key pressed), the display appears as shown in figure 5.9.

That was all pretty easy. But what if we want to be slightly pickier?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

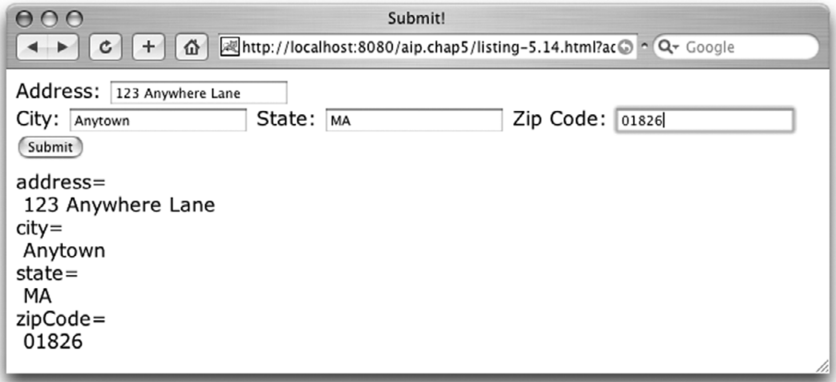

Figure 5.9 Submitting without submitting!

5.5.3 *Submitting only changed elements*

The previous example showed us that we can take control of the form-submission process and use event handling to reroute the submitted data to an Ajax request. Prototype's `serialize()` method made it almost trivial for us to gather all the data elements of a form to send to the server.

But what if we don't want to send *all* the form data? What if we only want to send data elements that have changed? Indeed, why make the request at all if none of the data has changed? We could use the `change` event of the form elements to know when an element's value has changed, but how do we best keep track of this information for use when it comes time to send the data to the server?

We could be sophomoric about it and store the information in global variables. But not only would that be inelegant, it would also create severe problems on pages with multiple forms, and is not an object-oriented approach.

We could be sophisticated about it and store the information right on the element itself by adding a custom property, as follows:

```
element.hasChanged = true;
```

We could then loop through the elements when it comes time to gather the data for submission, looking for elements that have this property set.

Or better yet, we can be clever about it (that sounds so much better than lazy) and leverage code that we already have handy. Listing 5.15 shows just such an approach.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 5.15 Submitting only changed data

```
<html>
  <head>
    <title>Submit, or not!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('infoForm','submit',submitMe);
        Event.observe('infoForm','change',        <-- 1 Establishes change
                      markChanged);                     handler on form
        $('addressField').focus();
      }
                                            2 Marks target element as changed
      function markChanged(event) {    <--
        Event.element(event).addClassName('changedField');
      }
                                          3 Collects only changed elements
      function submitMe(event) {    <--
        var changedElements = $$('.changedField');
        if (changedElements.length > 0 ) {
          var parameters = {};
          changedElements.each(
            function(element) {
              parameters[element.name] = element.value;
              element.removeClassName('changedField');
            }
          );
          new Ajax.Request(
            '/aip.chap5/handleSubmission',
            {
              method: 'post',
              parameters: parameters,
              onSuccess: function (transport) {
                $('info').innerHTML = transport.responseText;
              }
            }
          );
        }
        Event.stop(event);
      }
    </script>
  </head>

  <body>
    <form id="infoForm" action="/aip.chap5/shouldNotActivate">
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          <label>City:</label>
          <input type="text" id="cityField" name="city"/>
          <label>State:</label>
          <input type="text" id="stateField" name="state"/>
          <label>Zip Code:</label>
          <input type="text" id="zipCodeField" name="zipCode"/>
        </div>
        <div>
          <input type="submit" id="submitButton"/>
        </div>
      </form>
      <div id="info"></div>
    </body>
  </html>
```

In this example we've made some minor but significant changes to the code in listing 5.14. In the `onload` event handler, we've established a `change` event handler on the form ❶. We could have looped through the form, adding a handler on each individual element, but why bother when the form will receive the event notification during the bubble phase?

The handler function, `markChanged()` ❷, which will be called whenever a form element has changed, obtains a reference to the event's target element and adds the CSS class `changedField` to that element.

Huh? What does CSS have to do with keeping track of changed fields? All is revealed when we examine the changes to the `submitMe()` event-handler function.

In that function ❸, we use the Prototype `$$()` function. This handy function returns an array of all elements that match the CSS selector passed as its parameter. Since we specified the string `'.changedField'`, an array of all elements marked with that CSS class name is returned.

If that array is empty, we simply skip over the code that submits the request. Otherwise, we loop through the elements, creating an object hash of the name/value pairs that we gather from the array elements. That hash is then used as the parameter set for the Ajax request.

Since the data has been submitted and is no longer considered changed, we remove the CSS class name `changedField` from the elements, and we're good to go again!

Discussion

This example builds on the code in listing 5.14 to limit the parameters submitted on the Ajax request to those that have changed value, and to completely skip submitting the request if no changes have taken place.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

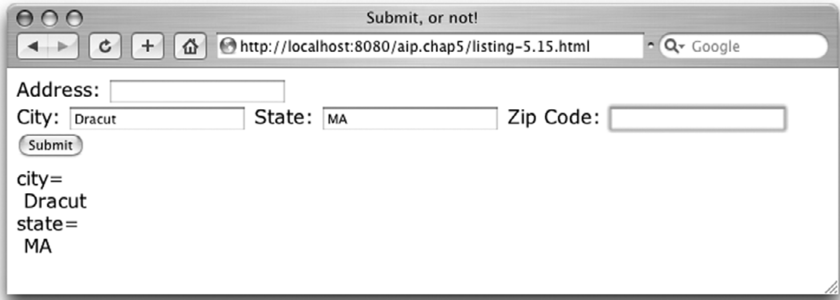

Figure 5.10 Submitting only what counts

We used a `change` event handler on the form to catch changes to all its elements, cleverly taking advantage of the bubble phase of event propagation. And we saw a clever way of marking elements for later identification through the use of CSS class names and the Prototype `$$()` function.

When displayed in the browser, and with only the City and State fields changed, we see the display as shown in figure 5.10.

5.6 Summary

In this chapter, we saw some interesting and powerful techniques to add interactivity to web applications. We looked at the various ways in which you can add event handlers to a DOM element, and we saw how the Prototype JavaScript library greatly simplifies the process of attaching and writing event handlers. We looked at all the major event types, and we examined many code snippets that demonstrated how these events can be used in our web applications. We also looked at some validation and form submission examples, something we'll cover more in-depth in the next chapter.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

6 Form validation and submission

This chapter covers

- Client-side field validation
- Client-side cross-field validation
- The POST HTTP request method
- XMLHttpRequest (XHR) form submissions

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Input validation is probably one of the least-favorite activities of web developers. "Why can't users just type their data in right the first time?" and "We're going to validate it on the server later anyway, so why bother?" are some common complaints, the latter having *perhaps* more merit. So why do it?

We do it because it creates a better user experience by

- Giving users immediate feedback regarding incorrect form data
- Cutting down on server resource usage (traffic and server cycles), which makes the user interface (UI) snappier

Input validation goes hand in hand with HTML forms: first, data is validated, and then it's submitted to the server. In the second part of this chapter, we'll discuss handling and submitting HTML forms as Ajax requests. Handling your own POST requests is in the lowest level of data handling, as you will be talking directly to the server yourself instead of letting the browser take care of it for you. This process can be quite error prone, but if implemented correctly, it can speed up your UI by eliminating those pesky browser refreshes.

6.1 Client-side validation

What is form validation? Simply put, it's a way of ensuring that the data a user enters into a form is valid. Now, *valid* is a bit of a nebulous concept, but basically it means that the data conforms to certain rules.

Form validation has been around for a while; first implemented mostly on the server side, it later moved toward the client side, which made for snappier feedback. Several server-side frameworks, such as Struts, are available, but sadly, client-side validation has always been a bit undersupported.

In this section, we'll take a look at creating our own extensible client-side validation framework. We'll build it up slowly with simple validations, and then move on to instant "as you type" validation and cross-field validation.

6.1.1 Validating on the client side

Validating on the client side can be a real… well, it's not fun. It sure would be great if you could just create a library that you could reuse on each page without having to rewrite it every time. And that's precisely what we'll be looking at.

As mentioned, we'll be using object-oriented JavaScript to ease both maintainability and extensibility. If you need a refresher, please refer to chapter 3 of this book.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

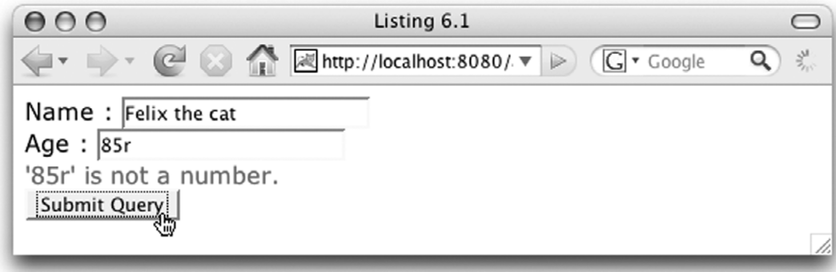

Figure 6.1 Validation in action

Problem

You need a reusable validation framework so that you don't have to rewrite client-side validation rules. It needs to be extensible, easy to maintain, and easy to use.

Solution

We'll use the assistance of the Prototype library to write our validation framework. Why Prototype? Because, as we saw in chapters 3 and 4, using a good JavaScript library makes it easier for us to write our validation framework in a more object-oriented way, which will promote code reuse. And everyone knows that object-oriented code is very easy to reuse! We'll use Prototype, but the same concepts can be applied using jQuery or the Dojo Toolkit.

Let's see what our framework can do with a look at figure 6.1. The user apparently slipped and typed an extraneous "r" character in a field that requires a numeric value. Now let's look at the HTML and script for this page in listing 6.1.

Listing 6.1 HTML for a validation framework

```
<html>
  <head>
    <title>Listing 6.1</title>
    <script type="text/javascript"
            src="prototype-1.5.1.js">      <- Uses Prototype
    </script>
    <script type="text/javascript"
            src="listing.6.2.js">      <- Contains validation framework
    </script>
    <script type="text/javascript">
      var framework = new ValidatorFramework();
      window.onload = function() {
        Event.observe('testForm','submit',
          function(event) {      <- Calls framework
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            if (!framework.validateForm(event.target))
              Event.stop(event);
          }
        );
      };
    </script>
    <title>Validation FrameWork</title>
    <style type="text/css">
      div.error {
        color: red;
      }
    </style>
  </head>
  <body>
    <form id="testForm"
          method="post"
          action="/aip.  chap6/requestInspector">
      <div> Name :
        <input name="name" type="text"
               id="name" valid="all"          <-- Marks input as validatable
               error="name_err"/>
        <div class="error" id="name_err"></div>
      </div>
      <div> Age :
        <input name="age" type="text"
               id="age" valid="number"        <-- Uses number validator
         error="age_err"/>
        <div class="error" id="age_err"></div>
      </div>
      <div>
        <input type="submit"/>
      </div>
    </form>
  </body>
</html>
```

Simple enough. We mark the fields that we wish to validate with two attributes in the input tag: `valid` tells the validator what validation rule to use; `error` tells it where to put any error message.

This technique of adding custom attributes to HTML elements is a useful but tricky approach because not all browsers support it in the same way. Internet Explorer and Safari make such attributes properties of the elements, but Firefox and other Gecko-based browsers do not. But all major browsers seem to allow retrieval of the value via the element's `getAttribute()` method. Now let's look at our validation framework classes (listing 6.2).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 6.2 Validation framework

```
var Validator = Class.create();

Validator.prototype = {
  type: "all",

  initialize: function(validators) {        // 1 Registers validator by its type
    validators[this.type] = this;
  },

  doValidate: function(input) {          // 2 Defines subclass hook
    return "";
  },

  validate: function(input, errordiv) {     // 3 Evaluates doValidate() output
    errorMsg = this.doValidate(input);
    errordiv.innerHTML = errorMsg;
    return (errorMsg.length == 0);
  }
}

var NumberValidator = Class.create();

Object.extend(NumberValidator.prototype,
              Validator.prototype);

Object.extend(NumberValidator.prototype, {   // 4 Takes care of all number fields
    type: "number",

     doValidate: function(input) {           // 5 Overrides doValidate() method
      var numberpattern=/(^\d+$)|(^\d+\.\d+$)/;
      if (numberpattern.test(input)) {
        return "";
      } else {
        return "'" + input + "' is not a number." ;
      }
    }
});

var ValidatorFramework = Class.create();

ValidatorFramework.prototype =
{
  validators: 0,                        // 6 Defines hash map of validators

  validateForm: function(form) {
    var retval = true;
    for(i = 0; i < form.length; i++) {
      currentInput = form[i];
      type = currentInput.getAttribute("valid");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        errorDivName = currentInput.getAttribute("error");
        if(type == null || errorDivName == null) {
          continue;
        } else {
          valid = this.validate(                         <-- 7 Validates with appropriate validator
            type, currentInput.value, $(errorDivName));
          if(!valid) {
            retval = false;
          }
        }
      }
      return retval;
    },

    validate: function(type, input, errordiv) {
      return this.validators[type].
        validate(input, errordiv);                       8 Grabs appropriate validator
    },

    initialize: function() {                             <-- 9 Sets up hash map, registers validators
      this.validators = new Object();
      new Validator(this.validators);
      new NumberValidator(this.validators);
    }

  }
```

Discussion

Let's talk about what's happening in listing 6.2. First, we'll discuss the Validator class. The Validator class will serve as the base class of all validators that we wish to implement. We've set its validation type to `all`, which does not mean much in this case; however, subclasses of Validator will set this to something more meaningful.

The ValidatorFramework class examines this property to decide which registered `Validator` implementation to execute when it reads the `valid` attribute of an `<input>` element. The ValidatorFramework instance is informed of this mapping in the `initialize()` method ❶ where the new Validator object registers itself. The real meat of the Validator class is its `doValidate()` method ❷, where we evaluate the input value. The Validator's `doValidate()` method returns an empty string, which is regarded as success. The `doValidate()` method is called from the `validate()` method ❸, which sets the validation error message, updates the error `<div>`'s contents to display the error message, and returns `true` or `false` depending on whether there were any errors.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Now that we've seen how the basic Validator superclass works, we'll take a look at how we subclass it (using the techniques we learned in chapter 3) to create our own Validator classes. We'll start with a simple NumberValidator class, which will validate all `number` fields ❹ with a regular expression in its overriding `doValidate()` method ❺. You can see how easy it is to create your own Validator classes: subclass Validator, set the appropriate type, and create an appropriate `doValidate()` method.

Our Validator classes are useless without something to drive them, which is where the ValidatorFramework class comes in. The ValidatorFramework class contains an object hash called `validators` containing all Validator objects that are registered with it ❻, which is set up in its `initialize()` method ❾. When we call its `validateForm()` method from our UI, it will look through all the input tags on the supplied form and find the appropriate validator ❼, ❽ for validating that input.

And what do we see if we run our example with invalid data? You can see for yourself in figure 6.1—85r is certainly not a number. Our validator works!

Frameworks such as these are easy to set up, especially using an object-oriented approach as we just did. You'll find such frameworks easy to maintain, as well as easy to extend—for example, with new validation rules. By using error `<div>`s and assigning them a CSS class, you can maintain a consistent look and feel across your entire application.

One caveat when dealing with client-side validation: *it is not secure!* We've mentioned this before, but it bears repeating. Even if you are validating the data on the client, you must *always* revalidate on the server. Crafty users can simply fake an HTTP POST to submit any data that they wish, totally bypassing your beautifully constructed validation framework.

You might also want to take a look back at chapter 5 to recall how to create server-assisted validations; such hybrid validations can cut down on your lines of code and ease the maintenance hassle associated with keeping server-side Java and JavaScript in sync.

A drawback of the framework we just implemented is that users must submit their form before they are notified of the validity of their inputs. A more user-friendly approach might be to immediately inform them of the errors of their ways. The following example will do exactly that by giving users instant feedback on the validity of their input.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

6.1.2 *Instant validation*

You've most certainly seen web applications that give you instant feedback on the validity of your inputs as you type. This saves users a lot of time and frustration, because they know that their inputs are valid before they click the submit button. This example extends our framework to support that type of behavior.

Problem

You wish to give users instant feedback on the validity of their inputs as they are typing.

Solution

It sure is nice when users can see whether they have made any mistakes in their data as they are typing it in, as opposed to filling in a form completely, only to find out when submitting that they have made errors.

We talked the big talk about code reuse throughout this book. As the authors do not wish to be accused of being "all hat and no cattle" (see http://en.wiktionary.org/wiki/all_hat_and_no_cattle), we'll reuse our validation framework to validate input fields as users are typing. In fact, the only thing we'll change is to add a `keyup` event handler to the form. First take a look at figure 6.2, which shows the error message delivered immediately after the "r" has been typed.

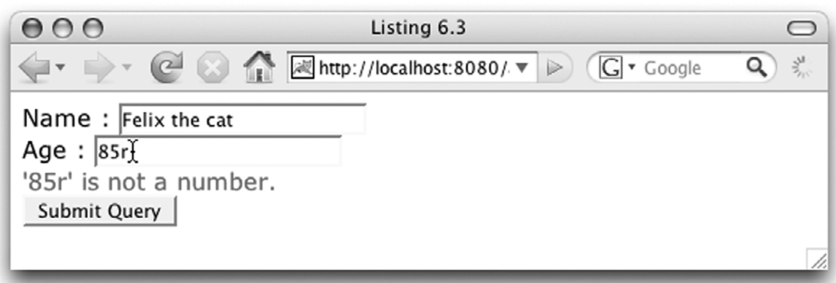

Figure 6.2 Instant feedback as you type

Rather than repeating the entire code of listing 6.1, listing 6.3 only shows the `onload` event handler, with the added code in bold.

Listing 6.3 As-you-type form validation

```
window.onload = function() {
  Event.observe('testForm','submit',function(event) {
    if (!framework.validateForm(event.target))
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        Event.stop(event);
    });
    Event.observe('testForm','keyup',function(event) {
      framework.validateForm(event.target.form);
    });
  };
```

Discussion

This solution didn't change very much from the solution of the previous section. However, this time we have constant feedback telling us when we type in a valid number.

But how does this work? You might be saying to yourself, "The `onkeyup` event handler is bound to the `<form>`, not to the `<input>` tags where I'm changing the data!" Good observation. Remember that in the browser event model, events propagate through the DOM tree. Therefore, if a `keyup` event is triggered on an `<input>` element, it will bubble up to the containing `<form>` element during the *bubble phase*, whose `onkeyup` event handler is invoked and calls our validator. There is no need to laboriously add `onkeyup` event handlers to each `<input>` element.

One annoying aspect of this approach is that, while we are typing into the name field, a message that the blank age field is invalid is displayed. How would you change our framework to alleviate this annoyance? (Hint: Consider using `onfocus` and `onblur` event handlers to enable and disable the instant validation when a field does not have focus.)

Another issue is that on each `keyup` event, validation for the entire form is performed needlessly. After all, we can only be changing one field at a time. How would you modify or extend the validation framework to allow individual field validation, and how would that help deal with the annoyance problem that we just mentioned?

Providing users with useful validations is not just an option these days; it has become a requirement. Users have become accustomed to easy-to-use web applications that inform them of errors at the first opportunity. Failure to provide them with a way to ensure the validity of their data before they incur the time expense of a server round-trip will result in complaints and lost customers.

Great—now we can validate single fields of data. But what if we have data being entered that needs to be validated in conjunction with the data from another field? This is what is called cross-field validation, and we'll tackle that topic next.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

6.1.3 Cross-field validation

Cross-field validation does not concern itself so much with whether individual element values are of the correct format (is it a number? is it an email address?), but rather that the values for two or more input elements are valid in relation to each other according to a set of business rules. For example, consider a form that asks for a start and end date of some event. Obviously, the start date must be before the end date. Our regular validators can take care of checking whether the entered values are valid dates, and if they are, we'll run the *cross-field validator* to make sure that they are also semantically correct—that the start date is before the end date. Let's get on with it.

Problem

You need to validate multiple fields in relation to each other.

Solution

Let's start developing our cross-field validation mechanism. We'll expand on what we've developed previously and add the capability to validate across fields. Let's begin with some HTML that sets up our form and cross-field validators (listing 6.4).

Listing 6.4 Cross-validation HTML

```
<html>
  <head>
    <title>Listing 6.4</title>
    <script type="text/javascript"
            src="prototype-1.5.1.js"></script>
    <script type="text/javascript"
            src="listing.6.5.js"></script>
    <script type="text/javascript">
      var framework = new ValidatorFramework();
      var xref1;
      var xref2;

      window.onload = function() {
        Event.observe('testForm','submit',function(event) {
          if (!framework.validateForm(event.target))
            Event.stop(event);
        });
        Event.observe('testForm','keyup',function(event) {
          framework.validateForm(event.target.form);
        });
        xref1 =                                  <-- Cross-validates first group
          new DateRangeCrossValidator(
            framework,
            new Array($('start'),$('end')),$('startend_err'));
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        xref2 =                                          <-- Cross-validates second group
          new DateRangeCrossValidator(
            framework,
            new Array($('start2'),$('end2')),$('startend_err2'));
      };
    </script>
    <style type="text/css">
      div.error {
        color: red;
      }
    </style>
  </head>
  <body>
    <form id="testForm"
          method="post"
          action="/aip.chap6/requestInspector">
      <div id="startend_err"                          <-- Specifies error <div>
           class="error"></div>
      <div> Start Date :
        <input name="start" type="text"               <-- Defines start date
               id="start" valid="date"
               error="start_err"/>
        <div class="error" id="start_err">            <-- Specifies another error <div>
        </div>
      </div>
      <div> End Date :
        <input name="end" type="text"                 <-- Defines end date
               id="end" valid="date"
               error="end_err"/>
        <div class="error" id="end_err"></div>
      </div>

      <div id="startend_err2"                         <-- Defines second group
           class="error"></div>
      <div> Start Date :
        <input name="start2" type="text" id="start2" valid="date"
               error="start_err2"/>
        <div class="error" id="start_err2"></div> </div>
      <div> End Date :
        <input name="end2" type="text" id="end2" valid="date"
               error="end_err2"/>
        <div class="error" id="end_err2"></div>
      </div>
      <div>
        <input type="submit"/>
      </div>
    </form>

  </body>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

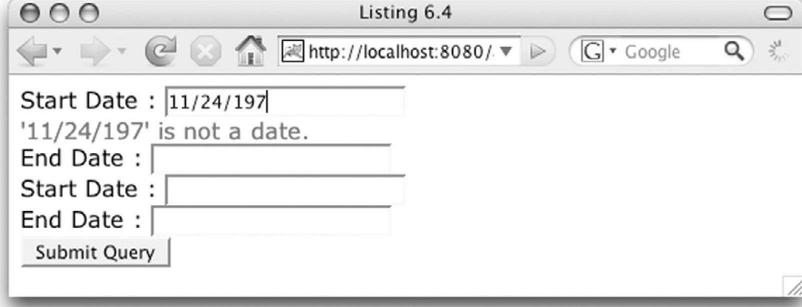

Figure 6.3 Validating an individual data field

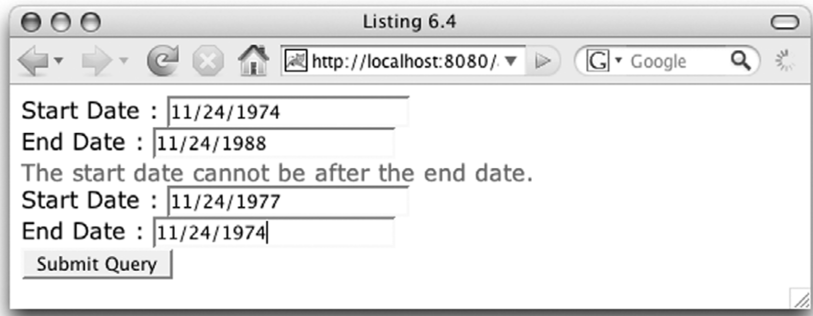

Figure 6.4 Cross-field validation in action

Basically, we defined error `<div>`s for each `<input>` element in combination with an error `<div>` for each cross-validation group where the error messages for that group should go. Then we construct our cross-field validators and pass them references to the `<input>` elements they must validate. Take a look at figures 6.3 and 6.4, and then review the code (listing 6.5) that makes all this happen. We made a few changes to the validation framework so that it can handle the cross-field validators.

Listing 6.5 The cross-validation framework

```
var Validator = Class.create();

Validator.prototype = {
  type: "all",

  initialize: function(validators) {
    validators[this.type] = this;
  },
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  doValidate: function(input) {
    return "";
  },

  validate: function(input, errordiv) {
    errorMsg = this.doValidate(input);
    errordiv.innerHTML = errorMsg;
    return (errorMsg.length == 0);
  }
}

var NumberValidator = Class.create();

Object.extend(NumberValidator.prototype,
              Validator.prototype);

Object.extend(NumberValidator.prototype, {
    type: "number",

     doValidate: function(input) {
      var numberpattern=/(^\d+$)|(^\d+\.\d+$)/;
      if (numberpattern.test(input)) {
        return "";
      } else {
        return "'" + input + "' is not a number." ;
      }
    }
});

var DateValidator = Class.create();        <-- ❶ Creates new class
Object.extend(DateValidator.prototype, Validator.prototype);
Object.extend(DateValidator.prototype, {
  type: "date",

   doValidate: function(input) {             ❷ Uses browser's date
    var value = Date.parse(input);       <-- object for parsing
    if(value <= 0) {
      return "'" + input + "' is not a date.";
    } else {
      return "";
    }
  }
});

var ValidatorFramework = Class.create();
ValidatorFramework.prototype =
{
  validators: 0,
  crossValidators: 0,     <-- Stores cross-field validators

  validateForm: function(form) {
    var retval = true;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    for(i = 0; i < form.length; i++) {
      currentInput = form[i];
      type = currentInput.getAttribute("valid");
      errorDivName = currentInput.getAttribute("error");
      if(type == null || errorDivName == null) {
        continue;
      } else {
        valid = this.validate(type, currentInput.value,
          $(errorDivName));
        if(!valid) {
          retval = false;
        }
      }
    }
    for(i = 0;                                      <-- 3 Iterates over validators
        i < this.crossValidators.length; i++) {
      this.crossValidators[i].clearErrors();
    }
    if (retval) {                                   <-- 4 Verifies validator completion
      for(i = 0; i < this.crossValidators.length; i++)  {
        valid = this.crossValidators[i].validate();
        if(!valid) {
          retval = false;
        }
      }
    }
    return retval;
  },

  validate: function(type, input, errordiv) {
    var validator = this.validators[type];
    if(!validator) {
      alert("No validator for type '" + type + "'.");
      return "";
    }
    return validator.validate(input, errordiv);
  },

  initialize: function() {
    this.validators = new Array();
    this.crossValidators = new Array();

    new Validator(this.validators);
    new NumberValidator(this.validators);
    new DateValidator(this.validators);
  }

}

var CrossValidator = Class.create();              <-- 5 Creates CrossValidator class
Object.extend(CrossValidator.prototype, {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    type: "none",
    crossError: 0,
    crossInputs: 0,

    initialize: function(framework,                  Defines
                         p_crossInputs,              initialize()
                         p_crossError) {             function
      framework.crossValidators.push(this);    <-- Registers with framework
      this.crossError = p_crossError;
      this.crossInputs = p_crossInputs;
    },

    validate: function() {
      errorMsg = this.doValidate(              Validates and sets any
        this.crossInputs);                 <-- error messages
      this.crossError.innerHTML = errorMsg;
      return (errorMsg.length == 0);
    },
                                            Clears out previously
    clearErrors: function() {          <-- generated errors
      this.crossError.innerHTML = "";
    }
});

var DateRangeCrossValidator =           6 Extends
  Class.create();                         CrossValidator class
Object.extend(DateRangeCrossValidator.prototype,
              CrossValidator.prototype);
Object.extend(DateRangeCrossValidator.prototype, {

  doValidate: function(inputs) {
    var startDate = Date.parse(inputs[0].value);
    var endDate = Date.parse(inputs[1].value);
    if (startDate > endDate) {
      return "The start date cannot be after the end date.";
    } else {
      return "";
    }
  }

});
```

Discussion

Let's look at what we just did in listing 6.5. We needed a Validator implementation that could handle dates, so we added one ❶ and implemented a proper `doValidate()` method ❷. We'll let the JavaScript Date class decide whether or not it is valid. Be careful, because the function will not work for dates before

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

January 1, 1970. If you need greater flexibility, you might want to consider using a regular expression.

We didn't need to change our framework too much. Just be aware that even though we do not cross-validate unless all the inputs have been validated, we do need to clear the errors that were generated by any previous cross-validation runs ❸. This is because the user might be entering data and could see nonsensical error messages. Once all the regular field validators pass ❹, we do a cross-field validation.

Because we're now doing cross-field validation, we create our base class for the cross-field validators ❺. This class looks a lot like the regular Validator class—its `initialize()` method now takes a reference to the framework it is associated with, along with an array of the `<input>` elements it is validating (the order is important here; look at the validator you are using to see what the order should be) as well as a reference to the `<div>` that will show any errors.

We've also subclassed the base cross-field validator ❻ to create a DateRangeCrossValidator class, which will check whether two dates are indeed in chronological order. This validator is passed a reference to the smaller date `<input>` field first and a reference to the larger date `<input>` field second.

And there you have it: a reusable and extensible cross-validation framework! Look back at figures 6.3 and 6.4 to see how this works in the browser. As you can see, no cross-field validation occurs until the values of all the `<input>` fields are correct, even though the second start/end combination is illegal. When we fix the problem with the first date (remember, our `DateValidator` won't deal with dates before 1970), we should see something like figure 6.4. The first combination is fine; the second one is not.

Of course, cross-field validation is not limited to start and end dates. The authors remember fondly an application that made good use of such cross-field validating techniques. The application in question was designed for the trading of electrical energy and, as such, relied on extensive knowledge of the capacities of the available power-generating facilities. There existed multiple interdependencies among the concurrent availability of power facilities, the amount of energy available in a given time block, and the amount of energy scheduled across an entire facility. We've given just a brief list of the interdependencies, but the concept is clear. Before the acceptance of a power schedule, it was important to ensure the validity of that schedule. The user interface took quite a while to draw, and a lot of data was submitted to the server. Needless to say, this took a long time to validate. To speed up the checking of the power schedules, we developed an extensive validation framework that could validate the entire schedule in a couple

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

of seconds. The old way of doing things could take up to two minutes. So cross-field validators are a good thing.

We now have our data validated and ready to ship to the server. But how will we do that? Good thing that the next section is there to show us the way!

6.2 Posting data

We saw in the previous chapter how to use Prototype to post a form via Ajax. In this section we'll take an in-depth look at how we can simulate the posting of a form ourselves through the XMLHttpRequest object (XHR). Because we can use the XHR object to make HTTP requests and specify what type of request to make, we can easily emulate a form submit by managing the data that we send to the server through the XHR.

Once we've shown how Ajax POST requests can be made, we'll take a look at using an alternative to Prototype, jQuery, to make such requests.

6.2.1 Anatomy of a POST

So what *does* a POST actually look like? Great question! Let's take a look, shall we? First we need a form (listing 6.6).

Listing 6.6 A simple form

```
<html>
  <body>
    <form method="post" action="http://localhost:2020/xyz">
      <input type="text" name="input1"/>
      <input type="text" name="input2"/>
      <input type="text" name="input3"/>
      <input type="submit"/>
    </form>
  </body>
</html>
```

All right, now we have a form. The `action` attribute may seem a bit confusing. Who is listening on port 2020? And the answer is… us! Using a utility called Netcat (http://netcat.sourceforge.net/), we can listen to arbitrary ports for incoming connections, which will be dumped to the command prompt. Netcat makes it easy to examine network data.

To run Netcat to listen on port 2020, we issue the following command line:

```
netcat -l -p2020
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

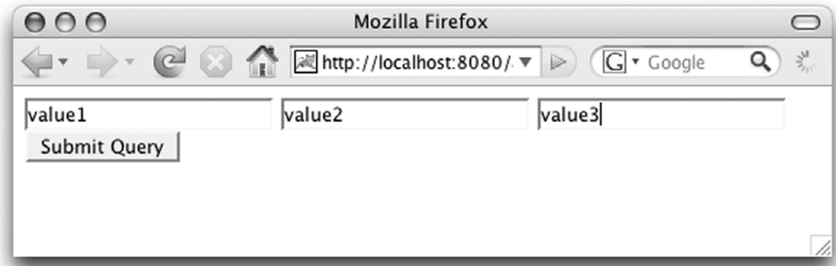

Figure 6.5 A simple form

Let's take a look at what the client posts to the server with the plain-as-vanilla form shown in figure 6.5.

When we submit this form, the data that is sent to port 2020 (which we are listening to via Netcat) might be as follows (there will be slight variations depending on the browser and its settings):

```
POST /xyz HTTP/1.1
Host: localhost:2020
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X;
   ➥ en-US; rv:1.8.1.2) Gecko/20070219 Firefox/2.0.0.2
Accept: text/xml,application/xml,application/xhtml+xml,
   ➥ text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

input1=value1&input2=value2&input3=value3
```

There are definitely a lot of interesting bits of information in there. For the sake of emulating a form post, we only need to specify some of those things.

The required first line contains the request method (in our case, POST), the requested resource, and the protocol being used. This is followed by a series of HTTP *headers*, consisting of key/value pairs separated by the colon character. The HTTP 1.1 protocol only requires that the `Host` header be sent. The remaining headers are optional but highly recommended.

The headers are followed by a blank line, which is in turn followed by the body of the POST request. For the content type that a form typically submits, the body consists of the URL-encoded values of the form data.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Looks a lot like the query string of a GET request, doesn't it? In fact, both a query string and POST body follow the same rules.

So, how do we perform a post with the XHR object? The following sections make this clear. Note that the first solution we'll consider doesn't take care of posting a form; we discuss that a bit later. The purpose of the upcoming solution is to show you how to post any sort of data.

6.2.2 Posting data to a server

Let's apply what we've learned in our analysis of POST requests earlier and show you how you can make your own POST request to a server via XHR. This is handy when you need to submit your own data to a server without forcing a browser refresh. You'll be able to post any data you want—XML, text, or whatever you have. You just need to construct the data you post appropriately and you're on your way.

Although you probably won't use this method frequently, it's often a good idea to have an understanding of how things work under the hood.

Problem

You need to post data to the server.

Solution

This solution is quite simple, and does not differ very much from previous direct uses of the XHR object. We simply obtain an XHR, set it up accordingly, perform a POST to a URL, and pass the data. Keep in mind that we are *not* emulating a POST the way that an HTML form would post the data; we'll handle that in the next solution. This solution can be used to post any data to the server.

Listing 6.7 contains our example code for emulating our own POST requests. One note about this code: it won't work in Internet Explorer 6. This is because, as we know, IE 6 uses an ActiveX object for XHR. We've already discussed how to create an XHR instance in a cross-browser manner, so in order to focus on the posting mechanism, we've simply assumed that the code would be run in Firefox, Safari, or Internet Explorer 7.

When we load it into a browser, we see two alerts pop up in succession. First, we see the client notification in figure 6.6, and once we click OK, the second alert box, figure 6.7, shows us the message we received from the server.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

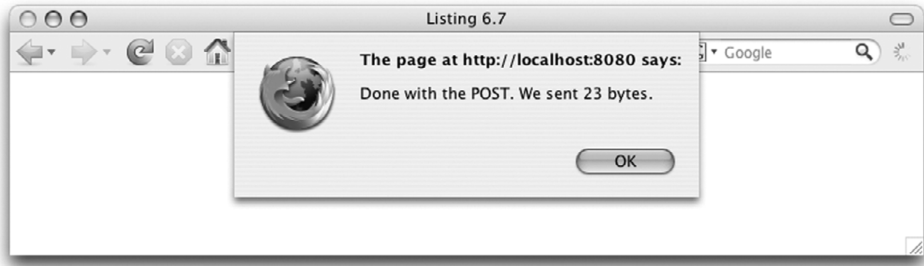

Figure 6.6 Client notification

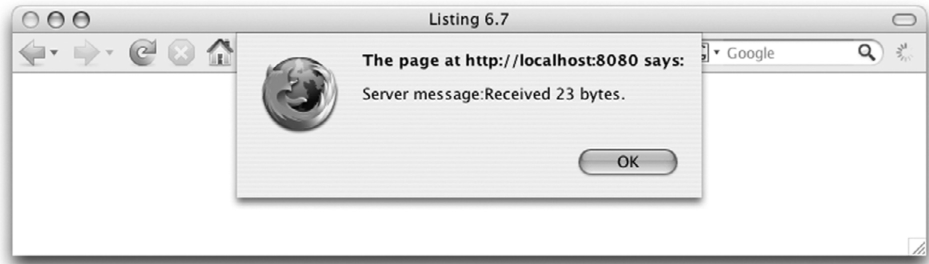

Figure 6.7 Server notification

Listing 6.7 Posting data to a server

```
<html>
  <head>
    <title>Listing 6.7</title>
    <script type="text/javascript">
      window.onload = function () {
        var data = "This is just some data.";
        var url = "/aip.chap6/postServlet";
        var xhr = new XMLHttpRequest();        <-- Creates XHR instance
        xhr.onreadystatechange = function () {
          if (xhr.readyState == 4 && xhr.status == 200) {
            alert('Done with the POST. We sent ' + data.length +
                  ' bytes.');
            alert('Server message:' + xhr.responseText);
          }
          else if (xhr.readyState == 4) {
            alert('Error posting. Server status: ' + xhr.status);
          }
        };
        xhr.open('POST',url,true);        <-- Specifies POST as method
        xhr.setRequestHeader(
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        "Content-Type", "whatever");          Specifies junk content type
      xhr.setRequestHeader(
        "Content-Length",data.length);        Specifies content length
      xhr.setRequestHeader(
        "Connection", "close");     Denotes a single request
      xhr.send(data);
    };
  </script>
</head>
<body>
</body>
</html>
```

We haven't really discussed the back end behind the post; let's do that now (listing 6.8).

Listing 6.8 POST-handling servlet

```
public class PostServlet extends HttpServlet {
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
      throws ServletException, IOException {
    System.out.println("Content type: " +
                       request.getContentType());
    ServletInputStream is = request.getInputStream();
    int data = is.read();
    int bytes = 0;
    while (data >= 0) {
      bytes++;
      data = is.read();
    }
    response.getWriter().write("Received " + bytes + " bytes.");
  }
}
```

This simple servlet merely gathers the request body and emits a response containing the length of the data received.

Discussion

It looks like posting data is pretty simple. Instead of using the HTTP GET method in the `xhr.open()` method, we simply specify `POST`. We can set the content type to whatever we want; if you are posting XML, you may wish to set it to `text/xml`. We also need to set the `Content-Length` request header. Even though the HTTP protocol may not strictly require it, this is important! If you make it too short, you can

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

lose data, because the server may assume that you're done sending it data even if you aren't. Likewise, if you make it too large, the server will hang, waiting for more data to arrive. After we've correctly determined the size of the data we're sending, we use the `xhr.send()` method to send the actual data.

Again, this approach looks simple, but it is quite powerful; you can use it to pass any type of data, such as XML or JSON, to the server. Because the server can also send back messages, you can use this mechanism as a type of remote procedure call: post some data, the server processes it, the client gets data back, and the client processes the data and updates the UI. This is how the IBM JavaScript SOAP/web services library works to exchange SOAP messages (see http://www-128.ibm.com/developerworks/web/library/ws-wsajax/).

Now you know how a post to the server is accomplished. Let's focus now on posting forms.

6.2.3 Posting form data to a server

Now that you have a grasp on the mechanics behind a POST request, let's take a look at posting a form to the server using Ajax. We were already introduced to just such an example (posting form elements without a page submit) in section 5.5.2, but where that example focused on the event-handling aspects, this section will examine the POST itself and we'll test things out with a wider variety of form elements.

Additionally, where the examples of the previous chapter used Prototype to handle making the request, this section will use another of the libraries that we explored in chapter 4: jQuery.

Problem

You need to emulate an HTML form post.

Solution

First, we'll set up a form with a variety of control elements to test whether the POST request is successfully submitting the values to the server. The blank form prior to any data entry looks like figure 6.8.

Then we'll set everything up so that "normal" form submissions are rerouted to submissions under Ajax control. Listing 6.9 shows the code. If you need a refresher on how jQuery works, now would be a good time to go review section 4.3.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

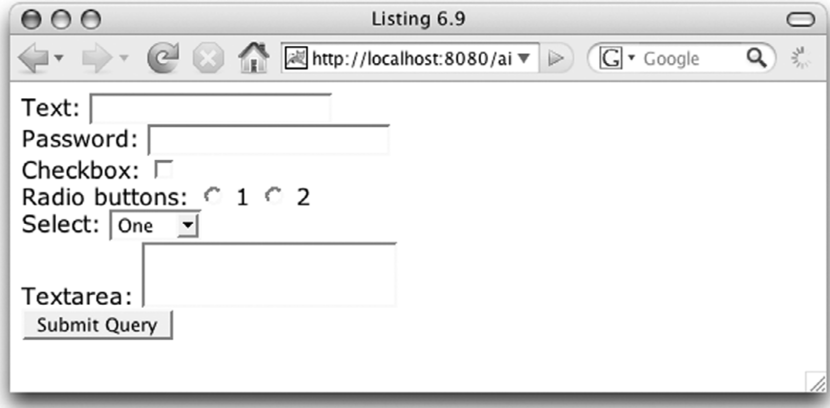

Figure 6.8 POST test form

Listing 6.9 Rerouting a form POST with jQuery

```
<html>
  <head>
    <title>Listing 6.9</title>
    <script type="text/javascript"                    ❶ Imports jQuery library
            src="jquery.js"></script>
    <script type="text/javascript"
            src="jquery.form.js"></script>            ❷ Imports jQuery form plug-in
    <script type="text/javascript">
      $(document).ready(function() {
        $('#testForm').ajaxForm({                     ❸ Prepares form for Ajax submission
          type: 'POST',
          target: '#results'
        });
      });
    </script>
  </head>

  <body>
    <form id="testForm"
          action="/aip.chap6/requestInspector">
      <div>
        Text:
        <input type="text" id="aTextField" name="aTextField"/>
      </div>
      <div>
        Password:
        <input type="password" id="aPassword" name="aPassword"/>
      </div>
      <div>
        Checkbox:
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <input type="checkbox" id="aCheckbox" name="aCheckbox"/>
      </div>
      <div>
        Radio buttons:
        <input type="radio" name="aRadioGroup" id="aRadio1"/> 1
        <input type="radio" name="aRadioGroup" id="aRadio2"/> 2
      </div>
      <div>
        Select:
        <select name="aSelect" id="aSelect">
          <option value="1">One</option>
          <option value="2">Two</option>
          <option value="3">Three</option>
        </select>
      </div>
      <div>
        Textarea:
        <textarea rows="2" name="aTextarea" id="aTextarea">
        </textarea>
      </div>
      <div><input type="submit"/></div>    <-- Container for
    </form>                                  4 server response
    <div id="results"></div>
  </body>
</html>
```

That seems simple enough. As you can see, using jQuery ❶ and its form plug-in ❷ made it so easy that it almost seems like cheating!

The `ajaxForm()` method ❸ does *not* submit the form. Rather, it prepares the form for submission under Ajax control when the form's `submit` event is eventually triggered. Without having to dig into the code for the plug-in, you can imagine how some of the steps that it needs to take are accomplished, given what you've learned so far in this book. Ponder how the `submit` event, event-handling mechanisms, and the XHR lesson of the previous solution can all be used to establish this functionality.

The options hash passed to the `ajaxForm()` method ❸ specifies a type of `POST` and a `target`, which is a DOM element in which the response will be displayed. This `target` option specifies an empty `<div>` element ❹ defined at the bottom of the page. Note how a CSS selector is used to identify the target element in the "jQuery way."

The server-side resource for this example is the same ParameterInspectorServlet class that we used in some of the examples in chapter 5. We won't go into the details of that class again, but recall that it gathers the parameters submitted to it and formats them for display.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

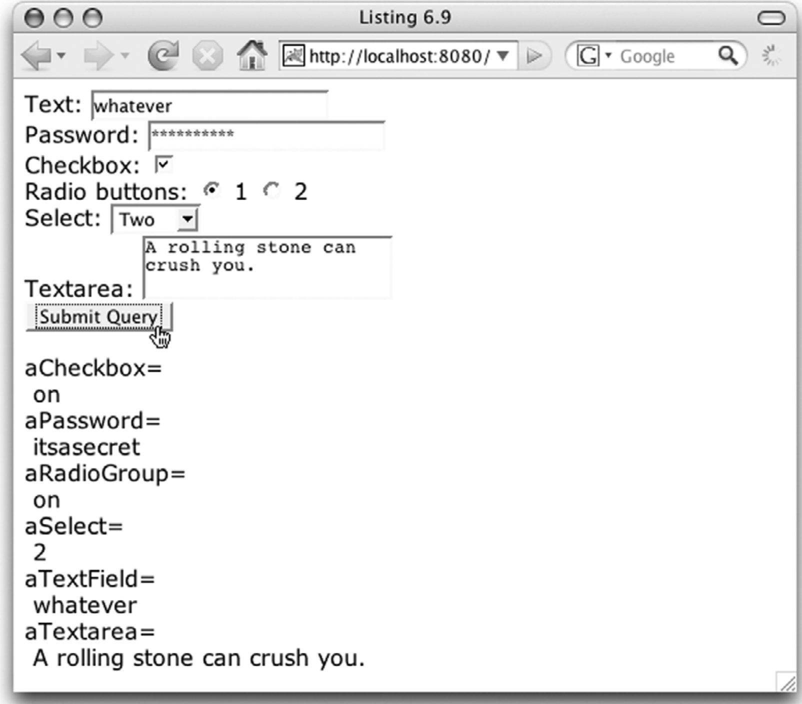

Figure 6.9 Form submission hijacked!

After filling in some data and submitting the form, our page appears as shown in figure 6.9.

Discussion

As you've seen, by "cheating" and using jQuery and its form plug-in, we've ensured that hijacking the form submission to channel it to the server via Ajax requires only a few lines of code in the page.

Emulating your own form posts in this way is a great win from a user perspective. The browser does not refresh, and a lot of time is saved not having to fetch the user interface (which probably has not changed) from the server. From a development perspective, server cycles and bandwidth are saved by not having to regenerate the user interface and send it back to the client. Naturally, you do need to develop client-side code to deal with the new way of doing things; mainly this will involve code that keeps the user interface in a coherent state now that we're not forcing a browser refresh.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

6.2.4 *Detecting form data changes*

To cut down on network traffic and database access, sometimes it behooves us to send across only those pieces of form data that have actually been modified in the form. We addressed this issue in section 5.5.3 with a focus on how to use event handling to accomplish this, but we gave short shrift to the details of dealing with actual form elements. In this section, we offer a more in-depth look at handling the form elements in an intelligent fashion. We'll also use jQuery just to give you another point of view with regard to event handling and DOM manipulation.

Problem

You want to detect whether changes have been made to form data and send the server only what has changed.

Solution

The way that we're going to solve this problem is quite similar to what we did in section 5.5.3. We'll set an `onchange` event handler on the `<form>` element that will apply a class name of `fieldChanged` to any control whose value changes, and establish an `onsubmit` event handler to intercept the form submission. But this time around, instead of just glossing over the nuances of dealing with the form elements, we'll pay more attention to what we're grabbing from the form to submit.

We'll take the solution of the previous example, with its plethora of input types, and augment it as appropriate. We'll also add a new control, a `<select>` element with `multiple` enabled, because that adds some complexity to our goal. When displayed, the page looks as shown in figure 6.10.

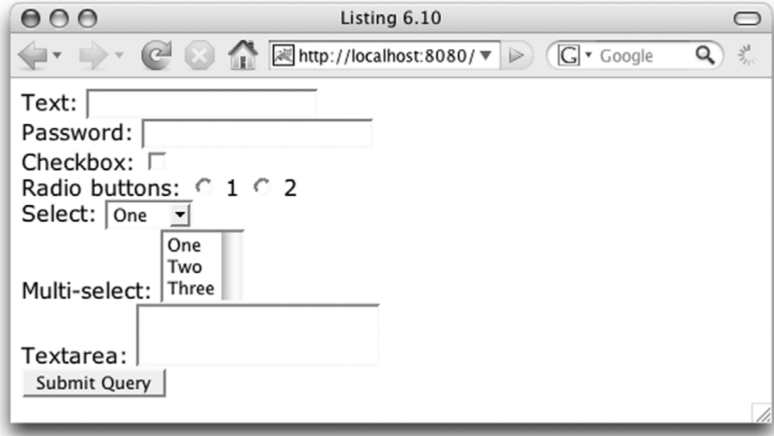

Figure 6.10 A cornucopia of waiting form elements

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Now let's look at the code for the page (listing 6.10). We've added quite a bit of script to handle the submission of only the changed elements.

Listing 6.10 Submitting only changed values

```
<html>
  <head>
    <title>Listing 6.10</title>
    <script type="text/javascript"
            src="jquery.js"></script>
    <script type="text/javascript">
      $(document).ready(function() {
        $('form').bind("change",
          function(event) {
            $(event.target).addClass('fieldChanged');
          }
        );
        $('form').bind("submit",
          function(event) {
            submitForm();
            return false;
          }
        )
        trackCheckboxes();
      });

      function trackCheckboxes() {
        $('input[@type=checkbox]').each(function() {

     $(this).bind('change',function() {return false;});
          var hidden = document.createElement('input');
          hidden.type = 'hidden';
          hidden.name = this.name;
          this.name = '_' + this.name;
          this.hidden = hidden;
          this.form.appendChild(hidden);
          $(this).bind('click', function() {
            var onOff = $(this).attr('checked') ? 'on' : 'off';
            this.hidden.value = onOff;
            $(this.hidden).addClass('fieldChanged');
          });
        });
      }

      function submitForm() {
        var params = {};
        $('#testForm .fieldChanged').each(function() {
          if (this.disabled) return;
          if (this.name.length==0) return;
          if ((this.type=='radio' || this.type=='checkbox') &&
```

1. Binds change event handler to form
2. Binds submit event handler to form
3. Tracks checkboxes
4. Handles form submission

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            !this.checked) return;
        if (this.type=='reset') return;
        if (this.type=='multiple' ||
            this.type=='select-multiple') {
          for(n = 0; n < this.length; n++) {
             if (this[n].selected)
               addParam(params,this.name,this[n].value);
          }
        }
        else {
          addParam(params,this.name,this.value);
        }
      });
      $.post(                                    ❺ Posts to server
        $('#testForm').get(0).action,
        params,
        function(data) {
          $('#results').empty().append(data);
        }
      );
      $('.fieldChanged')                          ❻ Restores state
        .removeClass('fieldChanged');
    }

    function addParam(params,name,value){        ❼ Collects parameters and values
      if (!params[name]) params[name] = new Array();
      params[name].push(value);
    }
  </script>
  <style>
    .fieldChanged {
      border: 1px solid red;
    }
  </style>
</head>

<body>
  <form id="testForm"
        action="/aip.chap6/requestInspector">
    <div>
      Text:
      <input type="text" id="aTextField" name="aTextField"/>
    </div>
    <div>
      Password:
      <input type="password" id="aPassword" name="aPassword"/>
    </div>
    <div>
      Checkbox:
      <input type="checkbox" id="aCheckbox" name="aCheckbox"/>
    </div>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <div>
          Radio buttons:
          <input type="radio" name="aRadioGroup" id="aRadio1"/> 1
          <input type="radio" name="aRadioGroup" id="aRadio2"/> 2
        </div>
        <div>
          Select:
          <select name="aSelect" id="aSelect">
            <option value="1">One</option>
            <option value="2">Two</option>
            <option value="3">Three</option>
          </select>
        </div>
        <div>
          Multi-select:
          <select name="aMultiSelect" id="aMultiSelect"
                  multiple="multiple" rows="3">
            <option value="1">One</option>
            <option value="2">Two</option>
            <option value="3">Three</option>
          </select>
        </div>
        <div>
          Textarea:
          <textarea rows="2" name="aTextarea" id="aTextarea">
          </textarea>
        </div>
        <div><input type="submit"/></div>
      </form>
      <div id="results"></div>
    </body>
  </html>
```

In the jQuery `ready()` handler for the document, we bind an `onchange` event handler to the `<form>` that gets triggered whenever a `change` event is fired for any of its contained elements ❶. This handler marks the field as changed by adding the `fieldChanged` class name to the field. In our example page, we added a CSS rule that draws a red border around such elements for diagnostic purposes. This helps us to visually check that the class name is being applied correctly while debugging the code. This is probably something you wouldn't keep in the code for final release (unless your requirements dictate informing the user which fields have changed).

We also apply an `onsubmit` event handler to the `<form>` element ❷ so that we can interrupt the normal flow of the submission and handle it ourselves.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Not much of that is new to us; we saw this in the example in section 5.5.3, but this time we've used jQuery rather than Prototype to establish the event handlers.

At the end of the `ready()` handler, we call a function named `trackCheckboxes()` ❸ that does something special with checkbox input elements. We're not quite ready to deal with that yet, so we'll defer talking about that until we understand the rest of the example.

The `onsubmit` event-handler function for the `<form>` element, `submitForm()` ❹, is where most of the interesting stuff is happening. In this function, we iterate through all the elements that have been marked as changed and build a list of parameters to submit in the `param` hash object. But notice how picky we're being!

We take great pains to reject any element that shouldn't be submitted. This includes disabled fields, those with empty names, radio or checkbox elements that aren't checked, and any reset element (that should never be submitted according to W3C rules).

We then collect the values from the elements that survive those tests. But it's more than just a matter of simple name and value pairs. Not only do we need to deal with multi-select elements, we also need to keep in mind that more than one control can have the same name. This means that for each parameter name, there can be more than one value. The `addParam()` function ❼ takes care of that by creating arrays in which to store the parameter values.

Finally, after all the changed element values have been collected, we use the `$.post()` jQuery function to post back to the server ❺ and remove the marking class name from all marked elements ❻.

As the last action in the sequence, we are sure to return `false` as the value of the `onsubmit` event handler to prevent the `<form>` element from continuing with its "normal" submission. After we play around with the example form and click submit, the page might look as shown in figure 6.11.

Now what about those pesky checkboxes?

Checkboxes pose a special challenge to change detection since they are rather unique among their input element brethren, in that checkbox elements do *not* get submitted as part of the HTTP request when they are not checked.

That means that without special consideration, we will have no way of reporting that a checkbox has changed from checked to unchecked state. So, we'll be clever and *give* these naughty elements special consideration! That is the purpose of the `trackCheckboxes()` ❸ function that, as you recall, we invoked as the very last act of the `ready()` handler.

The tactic that this function employs to track checkbox changes is to substitute a hidden field to represent the actual checkbox. This field will be given the value

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

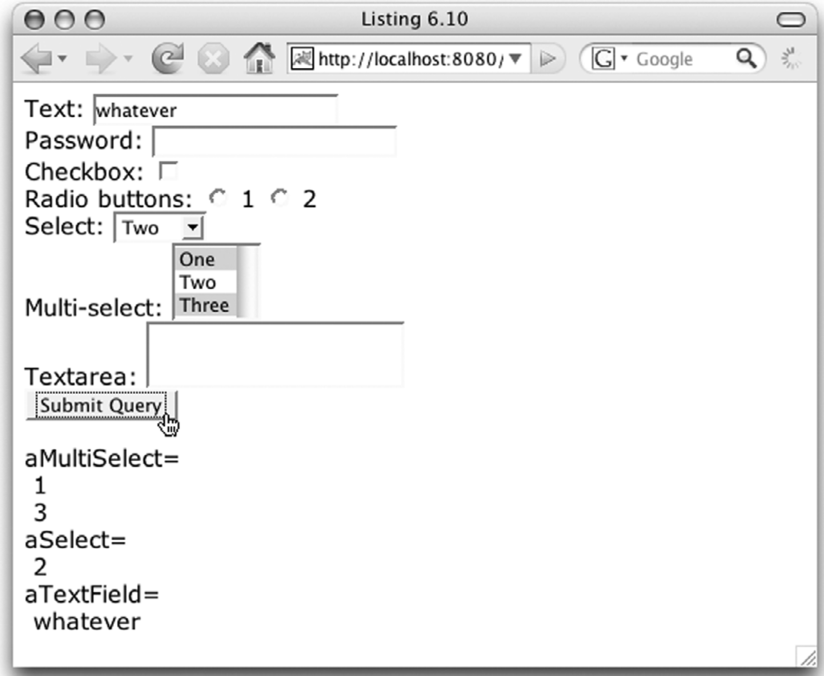

Figure 6.11 Only changed values have been submitted.

`on` or `off` depending on the state of the checkbox. That way, the value of this hidden doppelganger can be submitted when the state of the checkbox changes, even if the checkbox is unchecked.

The server-side code obviously needs to be prepared to deal with this change. Normally it would expect no parameter to be submitted on the part of an unchecked checkbox. With this tactic in place, a value of `off` will be passed for unchecked checkbox elements.

The function puts this plan into practice by iterating over each checkbox element (note how jQuery makes this easy), subverting the `onchange` handler for that element, and creates a new hidden element using the original name of the checkbox element. The name of the checkbox element is changed (rather arbitrarily, by prefixing it with an underscore) and a reference to the checkbox's hidden tracking element is set as a property of the checkbox. Finally, an `onclick` event handler is established on the checkbox that causes the hidden element's value to track the state of the actual checkbox element.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

While all that may seem a bit on the Byzantine side, it's not as complicated as it may seem at first, and it gives us the ability to reliably track checkbox changes in the same manner as the other input element types.

Discussion

In this solution, we've taken a deeper look at submitting only form values that have changed to the server via an Ajax POST request. We took more care this time to not do silly things like sending the values of disabled `<form>` elements. We also used jQuery to good advantage to take care of a lot of the mundane tasks, such as implementing event binding, adding and removing class names, and searching for DOM elements.

6.3 Summary

This chapter looked at a few concepts related to data handling with Ajax: validating and cross-validating your data before the post, and packing all that data up and sending it to the server by faking your own post. These techniques can lead to substantial speed increases in your application. Validating data as they type will save users from the hassle of waiting for the server to tell them that their data was incorrect. Performing Ajax form POSTs will save users from time wasted while the server is re-creating the page and performing a page refresh.

Remember that special care needs to be taken when faking a post, especially when it comes to the encoding of data and the setting of the appropriate headers for content length and type. Even then, you should thoroughly debug the code because it is easy to make a mistake. We strongly recommend you use libraries such as Prototype and jQuery; take advantage of the fact that someone has already done the hard work for you. Code reuse, as always, is the name of the game.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

7 Content navigation

This chapter covers

- Principles of content navigation
- Desktop and web influences
- Tab, window, and tree widgets
- Graceful degradation of JavaScript

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We can characterize most websites, rather simplistically, as consisting of a large repository of information residing on the server and a set of navigation mechanisms with which the user can retrieve subsets of this information in order to interact with it. In the case of a photo- or video-sharing site, the information being navigated consists of collections of pictures or other media. In a webmail application, the information is your mail. In an e-commerce store, the information is the catalog of goods, and so on. Whatever the site's nature and content, the underlying technical problem is the same: the server contains a lot of content, and users must be able to sift through it to find what they want. Ease of navigation is a strong differentiator between competing services, and it is a factor that any web-based application must address.

A lot has been written about the topic of content navigation and the Web, and various winning strategies have evolved and become standard features of many websites. We'll look at these briefly in this chapter, but only to set the stage for our main topic. Because this is a book about Ajax and its disruptive impact on the Web, we'll look at how Ajax has changed the field of web-based content navigation. We'll begin by looking at the problem of content navigation from a bird's-eye view and exploring the key factors to developing a navigation strategy.

7.1 Principles of website navigation

As we've said, the fundamental problem of many web-based sites is that the server contains a lot of information. The more successful our site is, the bigger this problem becomes. Let's suppose that your online store carries one hundred thousand different types of goods. Users are unable to process this volume of information, and they'll want to view different subsets of the data. At a technical level, this presents us with two problems: First, we need a mechanism for deriving the subsets of data that accurately meet each user's needs. Second, we need a way for users to interact with the site in order to express their needs as simply as possible. Let's take a look at each of these issues in turn.

7.1.1 Finding the needle in the haystack

The first problem that we face is finding the right data for our users. This is largely a back-end task, and this is largely a front-end book, but it's still worth spending a little time with it here in order to understand the principles. In very general terms, there are two ways that we can organize our information: categorization and classification. These are best explained by example, so let's pick a couple of examples that everyone should know.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Yahoo! employs categorization to make sense of the enormous data set that is the public Internet. Information is organized in a hierarchical set of categories, and each piece of information belongs in exactly one category: shopping, sport, finance, music, or whatever. A simple categorization might consist of only one level of categories, but Yahoo! employs a hierarchical categorization scheme. That is, within each category are further subcategories, thus creating a tree structure that can hold large amounts of information in manageable groups, no more than a few hops away from the top of the tree. Figure 7.1 illustrates the principles of categorization.

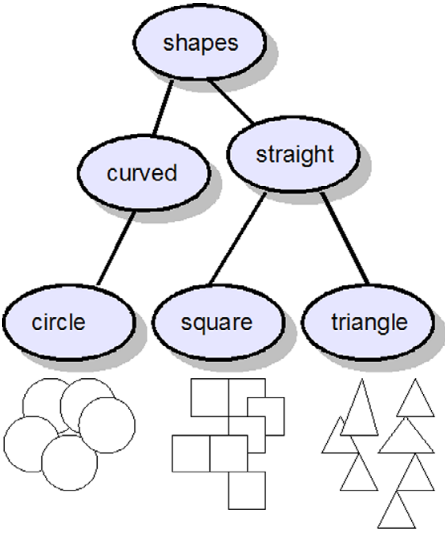

Figure 7.1 Categorization of data orders each element of information under exactly one category at any given level.

Classification differs from categorization in that any item of data may belong to more than one classification. If we think of Yahoo! as categorizing the Internet, then Google classifies it. The same web page may be returned by several unrelated search terms. In Google's case, the classification is generated automatically when the document is indexed by the spidering software and page-ranking algorithms. In other cases, classification data may be applied manually by the webmaster. Recently, the practice of allowing visitors to the site to classify content—commonly referred to as tagging—has come into vogue, and provides a powerful and scalable mechanism for organizing content on a site. Classification is depicted in figure 7.2.

Classification and categorization both have their strengths and weaknesses. Categorization doesn't scale as well as classification, with very large data sets eventually falling into one of three traps—too many categories at each level, too many levels of subcategory, or too many entries stored under each leaf node of the tree. However, categorization provides a reliable and more informative way for users to orient themselves by giving clues as to what they might find. An online store that simply presented a user with a search box and no hint as to whether it sold food, hardware, or clothing wouldn't attract many customers. In practice, most sites use a combination of the two techniques. Yahoo!, for example,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

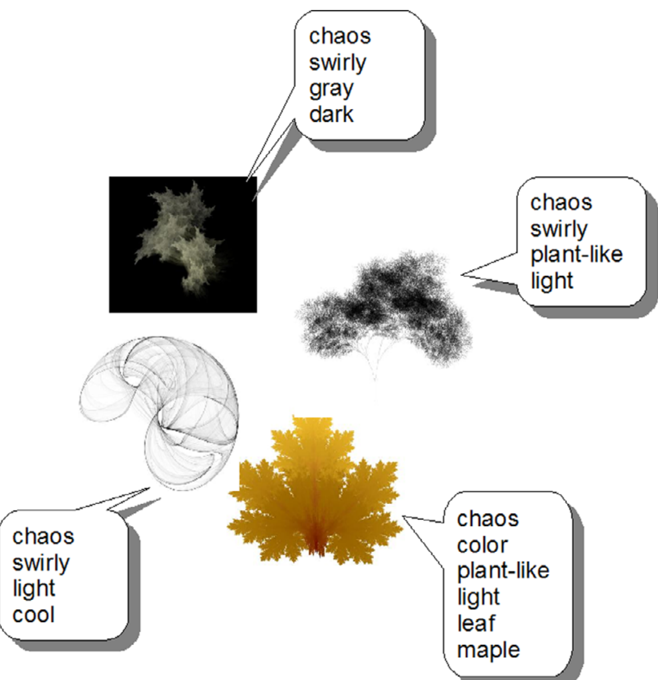

Figure 7.2 Classification does not organize the content on the server in a fixed pattern, but indexes or tags each element, thus allowing subsets to be created dynamically.

provides search facilities within its categories, and Google provides some top-level categories, such as web, image, and video search.

Now that we've examined the principles behind organizing our information, let's turn our attention to the user and see how to provide them with a way of driving our classification or categorization system.

7.1.2 Making a better needle-finder

We've established a means of organizing the content behind our site. Now we need to provide a way for users to use our organizational system to find what they want.

Categorization systems lend themselves to a number of user interface mechanisms on the Web. Popular solutions include sidebars, navigation bars, and breadcrumb trails. Yahoo! employs a simple sidebar at the top level of their site (see figure 7.3).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

If we look away from the Web to desktop applications, we can see that many common widgets are designed to deal with categorization systems, including menus, drop-down lists, tabs, and tree widgets. Some of these approaches have been adopted by websites, most notably hierarchical menus. Indeed, one of the few successful and widespread uses of Dynamic HTML (DHTML) before the advent of Ajax was in providing interactive drop-down menu bars that allow the presentation of many navigation options within a limited amount of space. Figure 7.4 shows an online store utilizing a number of navigation metaphors borrowed from the Web and the desktop.

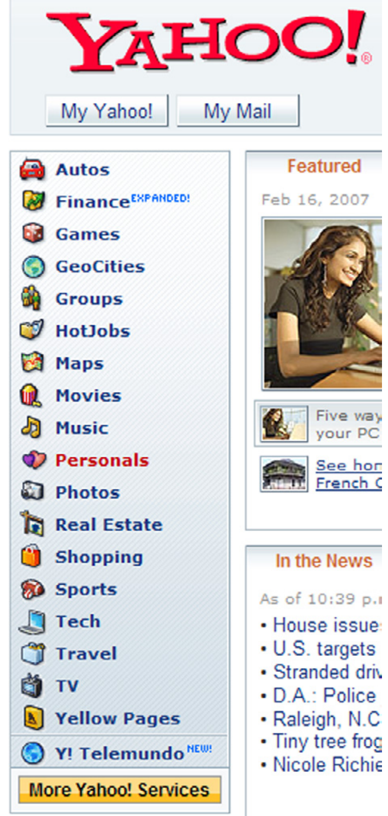

Figure 7.3 Yahoo!'s top-level categories are presented as a simple sidebar.

If we turn our attention to classification, we'll find that by far the most common navigation user interface is the humble search box, epitomized by Google's minimalist home page. Search can be enhanced by allowing additional fields, such as date ranges, or by mixing it with categorization schema, such as the Google family filter. With the advent of tagging, some alternative UI idioms have sprung up, such as the popular tags mechanism employed by Flickr (see figure 7.5).

All of these navigation systems can be supported by conventional hyperlinks and HTML forms, perhaps with a little bit of dynamic HTML to dress them up. As we said at the start of this chapter, our discussion so far involves simply setting the scene for a look at how Ajax has changed the field of web navigation by providing new mechanisms for assisting the user's navigation of a site's content. In the next section, we'll see what Ajax has brought to the party.

7.1.3 Navigation and Ajax

Ajax is a disruptive mix of technologies, and the positive impact of that disruption extends to the field of navigation. Unsurprisingly, the most important change it has brought about is the ability to improve the interactivity of navigation and to

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

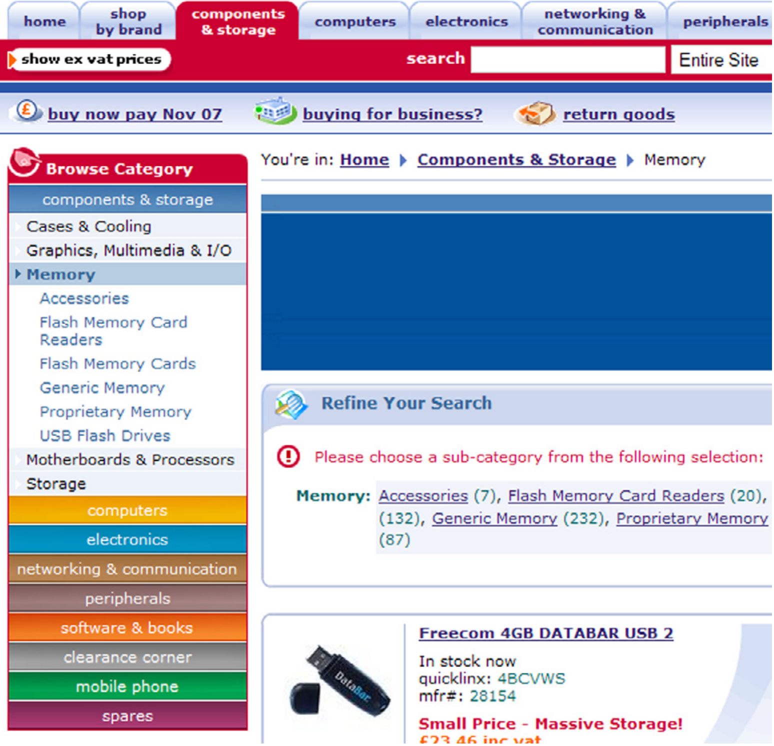

Figure 7.4 Online store dabs.com makes use of a number of UI widgets to orient the user within its complex categorization system, including a tree control (left), a set of tabs (top), and a breadcrumb trail (below the tabs).

provide better feedback to the user while they move around a site. Prior to Ajax, navigation widgets, whether composed of static HTML or DHTML, could only actually navigate to a new location by changing the URL of the current page, or the URL of a frame or IFrame on the page. Using Ajax, we can request information asynchronously from the server and update the page in a more incremental fashion. In a simple case, this can provide a more efficient route to updating a given region of the screen. At the more complex extreme, we may partially update several on-screen elements, creating a user experience much more like that of a desktop application.

If we consider navigation to be a two-stage process, in which interaction with the navigation widget is followed by retrieval of new content or data, then we can see that Ajax reaches the parts of the process that DHTML can't. DHTML navigation aids could improve the interactivity of the widget to the point of

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

All time most popular tags

06 africa amsterdam animals architecture art august australia autumn baby barcelona beach berlin birthday black blackandwhite blue boston bw california cameraphone camping canada canon car cat cats chicago china christmas church city clouds color concert d50 day dc december dog england europe fall family festival film florida flower flowers food france friends fun garden geotagged germany girl graffiti green halloween hawaii hiking holiday home honeymoon hongkong house india ireland island italy japan july june kids lake landscape light live london macro may me mexico mountain mountains museum music nature new newyork newyorkcity newzealand night nikon nyc ocean october paris park party people portrait red river roadtrip rock rome san sanfrancisco scotland sea seattle show sky snow spain spring street summer sun sunset sydney taiwan texas thailand tokyo toronto travel tree trees trip uk urban usa vacation vancouver washington water wedding white winter yellow york zoo

Figure 7.5 Flickr.com's alternative UI for browsing its classification system makes use of font size to indicate the popularity of items.

looking like a desktop app, but the retrieval of content was limited to presenting new content on the screen and therefore resembling a website rather than an application.

The current state of play with navigation in Ajax applications can be viewed as an exploration of the tensions between the user experience of a website and that of a desktop app. At one extreme, it is possible to create web pages that look and feel like desktop apps and use desktop-style GUI conventions throughout. At the other extreme, we can use Ajax to update regions of content, very much in the style of a web app. Between these two extremes are many interesting possibilities in which web- and desktop-style navigation combine to create a new kind of application.

We'll explore these possibilities throughout the remainder of this chapter. Let's start by looking at the traditional web-based approaches to navigation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

7.2 Traditional web-based navigation

As we've already said, an Ajax application can draw on navigation conventions both from the traditional Web and from desktop applications. With a conventional website as a starting point, the simplest transition that we can make is to stick with the web-based conventions for navigation but replace the hyperlinks or forms with asynchronous requests to the server. When we're working with a conventional web app and updating some features to incorporate Ajax, this is a good place to start.

We'll begin by looking at the case of a navigation menu and see what's involved in Ajax-enabling that.

7.2.1 A simple navigation menu

Let's take a look at what a simple navigation menu looks like when we're dealing with Ajax-style applications. As we mentioned earlier, we can't really use hard links; we'll need to use something a bit more advanced.

Problem

You are porting a standard website to a single-page application. Your navigation scheme needs to change to allow for linking to the content via an Ajax call.

Solution

For this example (see figure 7.6 for the result), we use a simple vertical list menu. When a menu item is selected, the desired content is retrieved and dynamically

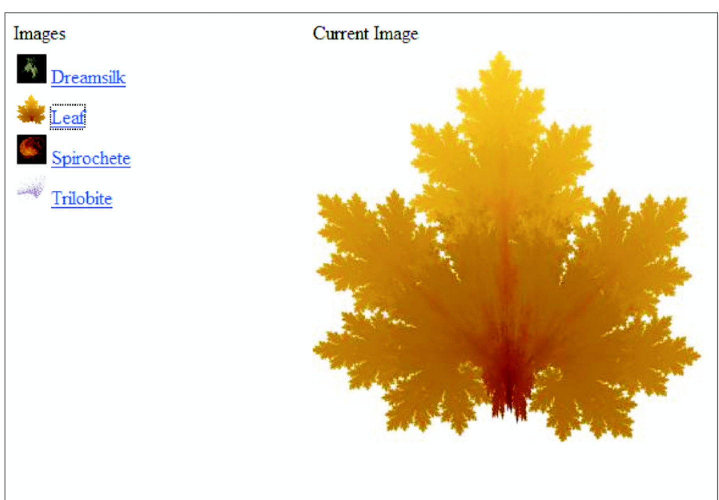

Figure 7.6 A simple navigation menu

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

loaded into the content area of the page. The first step is to create a function, `invokeLink(url)`, that requests the content from the server and inserts it into the DOM. After that, the links for each menu need to be changed to call the `invoke-Link()` function. It's as simple as that. Take a look at the code behind it (listing 7.1).

Listing 7.1 Our navigation menu

```
<html>
  <head>
    <title>Chaotic Images</title>
      <script type='text/javascript'
        src='../assets/js/jQuery.js'></script>          ❶ Imports jQuery library
      <script type='text/javascript'>
function invokeLink(url){
  $('#content_area').load(url);     ❷ Makes Ajax call
}
    </script>
  </head>
  <body>
    <table width='100%'>
      <tr>
        <td width='25%'>Images</td>
        <td width='75%'>Current Image</td>
      </tr>
      <tr>
        <td valign='TOP'>     ❸ Declares default content
          <table>
            <tr><td>
<img src='../assets/images/chaos/
  ➥ ifs/dreamsilk.jpg' width='24' height='24'>
<a href="javascript:invokeLink('dreamsilk.html');">Dreamsilk</a>
            </td></tr>
            <tr><td>
<img src='../assets/images/chaos/ifs/leaf.jpg' width='24' height='24'>
<a href="javascript:invokeLink('leaf.html');">Leaf</a>
            </td></tr>
            <tr><td>
<img src='../assets/images/chaos/
  ➥ ifs/spirochete.jpg' width='24' height='24'>
<a href="javascript:invokeLink('spirochete.html');">Spirochete</a>
            </td></tr>
            <tr><td>
<img src='../assets/images/chaos/
  ➥ ifs/trilobite.jpg' width='24' height='24'>
<a href="javascript:invokeLink('trilobite.html');">Trilobite</a>
            </td></tr>
          </table>
        </td>
        <td valign='TOP' id='content_area'>     ❹ Declares preview area
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            <br/><br/>
            Click on an image to the left to view it.
          </td>
        </tr>
      </table>
    </body>
  </html>
```

Discussion

We don't want to spend our time coding the Ajax request by hand, so we make use of a library to handle the low-level details for us. In this case, we've elected to use jQuery, which we load into the browser ❶ before we add our own script. Our `invokeLink()` function is trivially simple with jQuery there to help ❷. The `$()` function, as with the Prototype library, is used to select DOM elements. In jQuery, `$()` takes a CSS selector rule as argument. We've selected the target DOM node by its `id`, as defined in the HTML ❹. jQuery adds a `load()` method to DOM nodes resolved by the `$()` function, which will create an Ajax request for us and populate the node with the contents of the response. The HTML files that are loaded into the content area are actually HTML snippet files, which are merely an `<img>` tag pointing to the proper picture. The contents of these HTML snippets will replace the content area of the page, and the image is automatically displayed by the browser. These HTML snippets do not have to reflect actual files on the server; they could be server generated, allowing for a more dynamic UI.

All that remains is for us to attach the `invokeLink()` function to our UI. We do this here by using hyperlinks, such as anchor tags, with a JavaScript URL rather than an HTTP one, in the HTML that defines the menu ❸. This works but is simplistic, as we have to define the callback function as a piece of text rather than as code that can directly refer to variables elsewhere in our program. We'll look at how to programmatically attach events to the UI in the next example.

We've now explored how easy it is to directly request content from the server using an Ajax request. This was our very first, and our simplest, example. Let's move on to something that is a bit more complicated and that enjoys heavy use in web applications today: drop-down menus.

7.2.2 DHTML menus

Drop-down menus were one of the few success stories of DHTML, and a quick Google search will reveal the many variations available on the Internet for the would-be

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

web designer to pick up and customize. DHTML menus typically use JavaScript, and sometimes CSS, to control the interactivity of the menu widget, and then delegate the actions associated with active menu entries to a simple hyperlink.

In this section, we'll take one of the better examples available and modify it to use the `invokeLink()` function from our previous example to create a simple, interactive, Ajax-enabled menu.

Problem

You are modifying a website to make use of Ajax and run across a DHTML menu, which you need to Ajax-enable. Alternatively, you may have a new requirement to store a set of Ajax actions in categories. Either way, you'll find this widget useful!

Solution

The menu widget that we'll take as our starting point is based on the techniques described by Nick Rigby in his article "Drop-down Menus, Horizontal Style" (http://alistapart.com/articles/horizdropdowns). Nick describes how to create a hierarchical DHTML menu in which the majority of the layout and interaction is done using CSS rather than JavaScript. A small amount of JavaScript is added to overcome shortfalls in the CSS implementation in Internet Explorer.

The chaotic images that we used in our previous example all belong to a category called *iterated function systems* (IFSs). In this example, we've added two more images of a different type: Lindenmeyer systems, or L-systems. We've used Nick's approach to DHTML menus to organize our links into a hierarchical drop-down menu, DHTML style. We've also taken the opportunity to rewrite the JavaScript to make use of jQuery for greater elegance and simplicity. Figure 7.7 shows the menu in action; the code required to do this appears in listing 7.2.

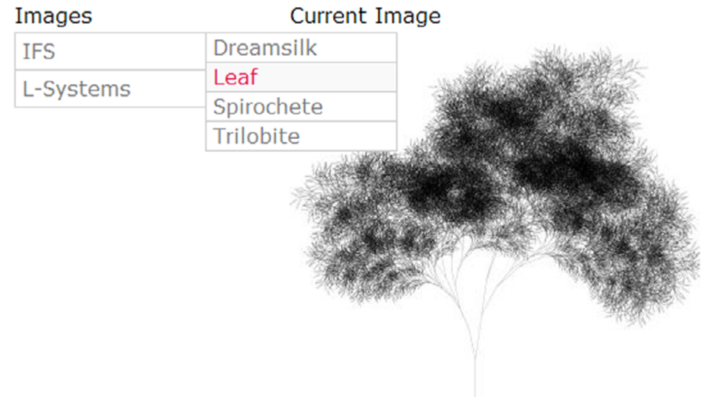

Figure 7.7 Simple DHTML menu enhanced with Ajax

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 7.2 DHTML navigation menu

```
<html>
<head>
<title>Chaotic Images</title>
<link rel="stylesheet" type="text/css"
  href="menu.css">                                    ❶ Imports CSS style sheet
<script type='text/javascript'
src='../assets/js/jQuery.js'></script>                ❷ Imports jQuery library
<script type='text/javascript'>
  function startList() {
    if ($.browser.msie) {                             ❸ Uses browser detection
      $("#nav > li").each(
        function(index, node){
          $(node).hover(                              ❹ Adds hover styles programmatically
            function() {
              $(this).addClass("over");
            },
            function() {
              $(this).removeClass("over");
            }
          );
        }
      );
    }
  }

  function invokeLink(url){                           ❺ Fetches content using Ajax
    $('#content_area').load(url);
  }

  $(startList);
</script>
</head>
<body>

  <table width='100%'>
    <tr>
      <td width='25%'>Images</td>
      <td width='75%'>Current Image</td>
    </tr>
    <tr>
      <td valign='TOP'>                               ❻ Declares menu contents
        <ul id="nav">
          <li><a href="#">IFS</a>
            <ul>
              <li><a
   href="javascript:invokeLink('dreamsilk.html');">Dreamsilk</a></li>
              <li><a href="javascript:
➥ invokeLink('leaf.html');">Leaf</a></li>
              <li><a href="javascript:
➥ invokeLink('spirochete.html');">Spirochete</a></li>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
                <li><a href="javascript:
                  ➥ invokeLink('trilobite.html');">Trilobite</a></li>
              </ul>
            </li>
            <li><a href="#">L-Systems</a>
              <ul>
                <li><a href="javascript:invokeLink('bush.html');">Bush
                    </a></li>
                <li><a href="javascript:invokeLink('weed.html');">Weed
                    </a></li>
              </ul>
            </li>

          </ul>
        </td>
        <td valign='TOP' id='content_area'>
          <br/><br/>
          Click on an image to the left to view it.
        </td>
      </tr>
    </table>
  </body>
  </html>
```

To make this example work, we need to import a CSS style sheet ❶ as well as the jQuery library ❷. Under Firefox, the CSS alone will make the menu behave, but under IE, we need to add some JavaScript. The CSS style sheet is available in the downloadable source code for the examples in this chapter at www.manning.com/crane2, and is unmodified from Nick Rigby's original code, so we won't spend time on it here. We're not interested in how the DHTML works, but in adding Ajax.

Before we consider that, though, let's take another look at what jQuery can do for us. In the function `startList()`, we've used several features from jQuery that are worth noting. The first thing to note is that jQuery has made browser detection cleaner for us than if we were to manually inspect the user-agent string ❸.

Again, we've used `$()` with a CSS selector as argument. This time, the selector will match more than one DOM element, and so `$()` will return an object that wraps an array. We can use the `each()` method to iterate over this array. `$.each()` accepts an iterator function as argument (that is, the function that will be applied to each element in the array). In our iterator, we add event handlers on `mouseover` and `mouseout` to alter the CSS of the elements ❹. The `hover()` method that jQuery adds to the DOM element makes this extremely easy for us. We also make use of the `addClass()` and `removeClass()` methods, which allow us to modify CSS classes with a fine degree of control.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Finally, we invoke `$()` again, this time with the `startList()` function as argument. When invoked with a function, jQuery's `$()` will bind the function as a listener to `window.onload`, invoking it when the DOM for the page is fully loaded. We could have simply written

```
window.onload=startList;
```

but using `$()` has the advantage of allowing us to add more than one listener to `load`. That's no big deal in an example of this size, but it's very useful when writing larger, modular systems.

So, that's the menu sorted out. To add Ajax, we can simply provide our jQuery-powered `invokeLink()` function again ❺ and add hyperlinks to the menu nodes. The menu is declared as a set of HTML unordered list elements ❻, whose appearance is modified by the CSS and optional JavaScript. And so, when the page is loaded, our menu springs to life.

That concludes this section, in which we looked at web-style navigation. In the next section, we'll explore some examples that examine the other end of the spectrum: the desktop application approach to user interfaces.

7.3 Borrowing navigational aids from the desktop app

While the Web was still in its infancy, desktop application developers had long been wrestling with navigation issues and had developed a number of conventions for organizing visual content. The look and feel of the DHTML menus that we looked at in the previous example is largely borrowed from desktop applications, but by and large, desktop and web applications have had little direct overlap. One of the reasons for this is that traditional web apps have been tied to the full-page refresh model, whereas desktop UI conventions have evolved based on the possibility of incremental updates. When interacting with complex UI elements such as trees, grids, and toolbars, each interaction will typically only modify a small part of the screen.

Ajax has changed this situation. By allowing incremental updates from the server, it has created a better fit with desktop navigation techniques, and web apps that look and feel like desktop applications are starting to emerge.

We stated at the beginning of this chapter that desktop look-alikes represent one extreme of Ajax development, and they are certainly not a logical conclusion for all web-based applications. There are plenty of interesting approaches to navigation going on somewhere in between the desktop and traditional web models, and we'll return to that middle ground later in this chapter. For now, though, let's

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

consider what the world of the desktop application has to lend us in terms of navigation support.

7.3.1 *The qooxdoo tab view*

In the previous example, we used CSS and a bit of JavaScript to add some behavior to a piece of HTML that we had declared in the body of the page. The qooxdoo widget library takes a radically different approach to authoring a web UI, helping you avoid many of the pitfalls in web UI development by presenting you with an object-oriented (OO) JavaScript API for creating and manipulating UI components. If you have used GTK, AWT, Swing, or another thick-client OO GUI toolkit, you'll feel at home here.

Problem

Your application contains a group of several pieces of discrete content but you want only one group shown at a time. These pieces of content could be many things: navigations, forms, and so on.

Solution

The first thing you need to know about qooxdoo is that initialization of the application needs to take place in the `window.application.main()` function. You can do this by providing an anonymous implementation of this function, as you can see in listing 7.3 in a moment. You can see the result in figure 7.8.

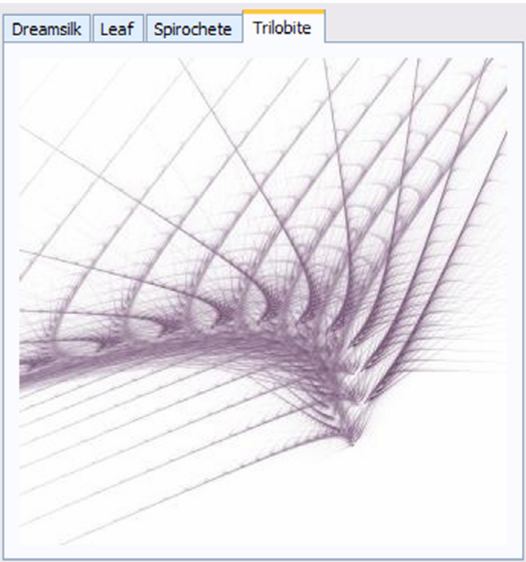

Figure 7.8 The qooxdoo tab view

In setting up our tab view, the first thing we need to do is create an instance of QxTabView, and then set the left, top, width, and height. From there we can create our four QxTabViewButton instances (which will provide a means for toggling the tabs), add them to the tab view instances bar container, and set the first one to be selected. Next, we need to create the actual QxTabViewPage instances and add those to our tab view instances pane container. Then we create

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

a QxImage instance for each of our images and add them to their corresponding pages. Finally, we add the tab view instance to the client document, qooxdoo's root-level container. Everything in qooxdoo lives within the client document.

You should now be able to fire up the example and tab away to your heart's content!

Listing 7.3 Using the qooxdoo tab view

```
<html>
  <head>
    <title>Chaotic Images</title>
    <script src='../assets/js/qooxdoo/include.js'
            type='text/javascript'></script>

    <script LANGUAGE='JavaScript'>
      <!--
  window.application.main = function()
  {
    var tf1 = new QxTabView();                     ❶ Declares new tab view
    tf1.set({ left: 20, top: 48, width: 342, height: 362 });

    var t1 = new QxTabViewButton('Dreamsilk');     ❷ Creates tab view buttons
    var t2 = new QxTabViewButton('Leaf');
    var t3 = new QxTabViewButton('Spirochete');
    var t4 = new QxTabViewButton('Trilobite');

    t1.setChecked(true);

    tf1.getBar().add(t1, t2, t3, t4);              ❸ Adds buttons to tab bar

    var p1 = new QxTabViewPage(t1);                ❹ Creates tab view pages
    var p2 = new QxTabViewPage(t2);
    var p3 = new QxTabViewPage(t3);
    var p4 = new QxTabViewPage(t4);

    tf1.getPane().add(p1, p2, p3, p4);             ❺ Adds pages to view pane

    var i1 = new QxImage('../assets/images/        ❻ Creates image objects
      ➥ chaos/ifs/dreamsilk.jpg');
    var i2 = new QxImage('../assets/images/chaos/ifs/leaf.jpg');
    var i3 = new QxImage('../assets/images/chaos/ifs/spirochete.jpg');
    var i4 = new QxImage('../assets/images/chaos/ifs/trilobite.jpg');

    p1.add(i1);                                    ❼ Adds image objects
    p2.add(i2);                                       to pages
    p3.add(i3);
    p4.add(i4);

    this.getClientWindow()
      .getClientDocument()
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        .add(tf1);     <-- 8 Adds tab view to client document
    };
        //-->
      </script>
    </head>
    <body>

    </body>
  </HTML>
```

Discussion

In listing 7.3, we first declare a new tab view object ❶. After that, we create four new tab view button objects, and set the first one to be selected ❷. We add these newly created tab view buttons to the tab view bar ❸. We also need to create four new tab view page objects ❹, which will hold the actual contents of the tab, and add them to the tab view's pane ❺. We haven't yet supplied our content. To that effect, we create four new image objects ❻ and add each image to its respective page ❼. Finally, we add the tab view to the client document ❽ for it to be displayed.

This first introduction to qooxdoo's API gives you just a taste of how simple it is to configure its widgets and how very different it is from writing a conventional web application. Very little JavaScript and no HTML (note that the body tag is completely devoid of content) go a long way here. There is one major drawback to using qooxdoo and that can also be argued as its greatest strength: it completely destroys the web content design/layout model. A graphic designer can no longer go into a WYSIWYG editor and lay out a UI.

The tabbed panel gave us a taste of what qooxdoo can do, but we addressed a problem that could be tackled in a more conventional development approach, as we'll see with the Rico Accordion later in this chapter. In the next example, we'll raise the stakes and throw in some more distinctly desktop metaphors to see how far we can push this style of development.

7.3.2 The qooxdoo toolbar and windows

Using qooxdoo again, we'll now explore the windowing and toolbar features this toolkit provides. Dividing the screen real estate into windows provides a flexible alternative to the tabbed pane, and will allow users to view more than one resource side by side if they so wish.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

You want to provide your users with greater control over how they allocate screen space between several regions of content by providing a multiple-document interface.

Solution

We can meet this requirement by providing users with a toolbar that will allow them to launch each resource as a window control. Using qooxdoo, the toolbar and window controls are ready-made for us, and we simply need to assemble them. Figure 7.9 shows the finished result.

Achieving this is not too difficult, as qooxdoo does most of the heavy work for us. Listing 7.4 shows what's needed.

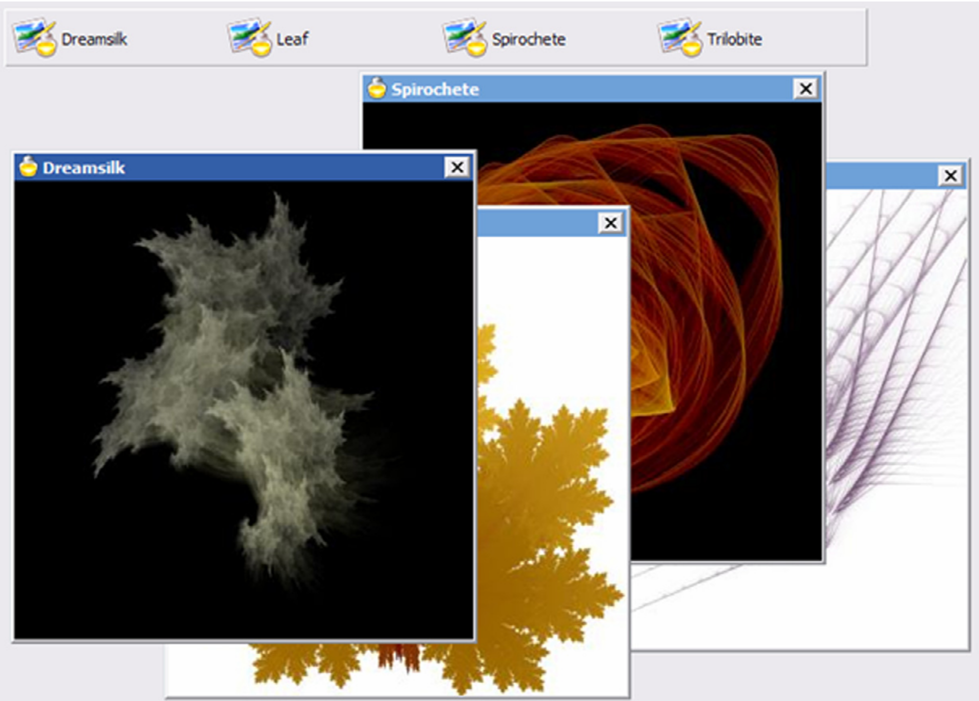

Figure 7.9 The qooxdoo toolbar and windows

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 7.4 The qooxdoo toolbar and windows example

```
<html>
  <head>
    <title>Chaotic Images</title>
    <script src='../assets/js/qooxdoo/include.js' type='text/javascript'></
   script>

    <script type='text/javascript'>
      <!--
window.application.main = function(){

  var d  = this.getClientWindow()
    .getClientDocument();
  var tb = new QxToolBar();                            ❶ Creates toolbar object
  tb.set(
   {top : 20, left : 20, width : 602}
  );

  createLaunchButton(tb, 'Dreamsilk',
    '../assets/images/chaos/ifs/dreamsilk.jpg');
  createLaunchButton(tb, 'Leaf',
    '../assets/images/chaos/ifs/leaf.jpg');            ❷ Adds buttons to toolbar
  createLaunchButton(tb, 'Spirochete' ,
    '../assets/images/chaos/ifs/spirochete.jpg');
  createLaunchButton(tb, 'Trilobite',
    '../assets/images/chaos/ifs/trilobite.jpg');

  d.add(tb);
};

var windowCount=0;

function createLaunchButton(toolbar,title,image){
  var button = new QxToolBarButton(                    ❸ Creates button object
    title, 'icons/32/bitmapgraphics.png'
  );
  button.setWidth(150);
  button.addEventListener(                             ❹ Adds event handler
    'execute',
    function(){
      if (!button.window){
        var d  = window.application
          .getClientWindow()
          .getClientDocument();
        var win=new QxWindow(                          ❺ Creates window object on demand
          title, 'icons/16/bitmapgraphics.png'
        );

        win.setSpace(
          20+(48*(windowCount+1)), 320,
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      20+(48*(windowCount+1)), 320
    );
    win.set(
      {showMinimize : false,
       showMaximize : false,
       resizeable : false}
    );
    win.add(new QxImage(image));

    d.add(win);

    button.window=win;
    windowCount++;
  }
  if (button.window.isSeeable()){
    button.window.close();
  }else{
    button.window.open();
  }
 }
);

toolbar.add(button);
}

      //-->
    </script>
  </head>
  <body>
  </body>
</html>
```

❻ **Hides or shows window**

As before, our HTML page contains no HTML markup in the body, as qooxdoo will generate all the DOM elements for us. We assemble the widgets in the `window.application.main()` method, as before.

To create the toolbar, we need to invoke the constructor for the toolbar object itself ❶, and then add each of the buttons to it ❷. Adding the buttons will require a few extra steps, so we have pulled that out into a helper function called `createLaunchButton()`.

Within this function, we create the qooxdoo button object ❸, and then add an event handler to it ❹. When the button is clicked, we want it to toggle between showing and hiding the window ❻, but we first need to ensure that the window has been created. We can do that by assigning a new property called `window` to each button that will be either `null` or the qooxdoo window object. So, before hiding or showing, we first check whether this property is set, and, if it's

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

not, we create the qooxdoo window object on demand ❺. This part of the code will be invoked only once, when the button is first clicked.

Note that we've defined the event handler inline as an anonymous function, allowing us to create a closure on the button object.

Discussion

This example was a simple introduction into the power of using a multiple-document interface within a browser window. If you are purely a web developer and have never been exposed to traditional thick-client OO GUI design, qooxdoo may seem a bit obtuse and counterintuitive. However, the way in which it abstracts all of the dirtiness of HTML and presents you with a clean, concise OO API for web applications is a pretty powerful thing.

It is also necessary to consider your audience when developing an interface of this type. The multidocument interface gives the user a lot more control over the layout of the page, but also asks a lot more of them. Users don't necessarily want more control over every aspect of the user interface, and your judgment as to the needs of your audience is critical here. In chapter 1, we discussed the distinction between line-of-business applications and those intended for casual use. We suggest that the type of "power-user" interface that we're seeing here might be suitable for line-of-business applications, in which a user is willing to invest time and effort in configuring the layout, but not for a casual-use application such as a shopping cart or a dictionary.

Used in the right place, then, a framework such as qooxdoo can be invaluable. Before we finish with qooxdoo, we'll take it for one final spin, this time setting it to work on one of the most complex of UI widgets: the tree.

7.3.3 The qooxdoo tree widget

Tree widgets are among the most sophisticated and powerful navigation widgets in common use. Although simpler controls such as tabs and menus have seen considerable adoption on the Web, trees have not generally been taken up with enthusiasm, possibly because the effort required to interact with them is poorly suited to the casual-use application.

Ajax brings line-of-business applications within reach of the web application, though, and so we can expect an increasing demand for more sophisticated controls such as trees. With its sights set on exactly this target, qooxdoo provides us with a ready-to-go tree widget, and in this example, we'll see how to make it work for us.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

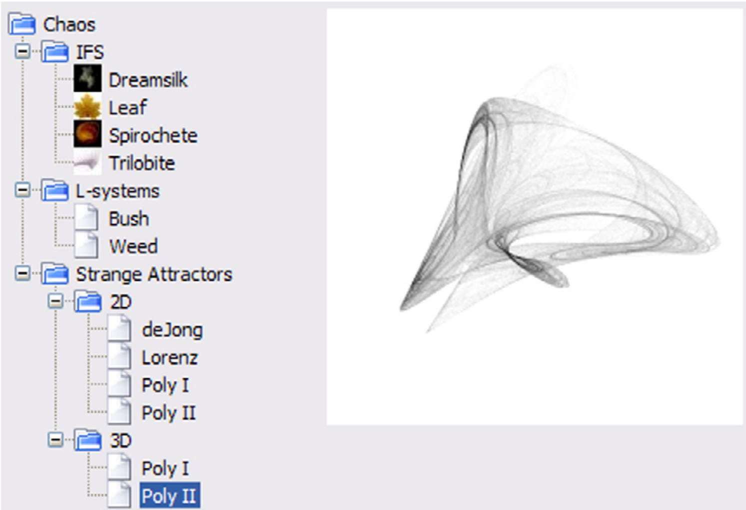

Figure 7.10 The qooxdoo tree control

Problem

We want to present users with a data set that is divided into many categories and subcategories, without overloading them with hundreds of options at once.

Solution

Use a tree widget! Tree widgets are complex, and we don't want to get bogged down in looking after node event handlers, drawing lines connecting the nodes, and other such implementation details ourselves. The qooxdoo library provides a tree widget control that will allow us to focus on the business at hand: organizing our data and presenting it to the user. For this example, we've added a third set of fractal images, belonging to the set known as "Strange Attractors." Within this set, we've subdivided our images into images of two- and three-dimensional chaotic patterns. Figure 7.10 shows how the application looks, but because of the limits of modern print technology you'll have to download and run the example yourself to see the 3D images animate. Listing 7.5 shows how we set up the tree control.

Listing 7.5 Using the qooxdoo tree widget

```
<html>
  <head>
    <title>Chaotic Images - Tree Navigation</title>
    <script src='../assets/js/qooxdoo/include.js'
      type='text/javascript'></script>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      <script type='text/javascript'>
        <!--
    window.application.main = function()
    {
      var d = this.getClientWindow()
        .getClientDocument();

      var panel = new QxImage(          ❶ Creates preview panel
        '../assets/images/chaos/ifs/spirochete.jpg'
      );
      panel.set({
        left: 204, top: 48,
        width: 320, height: 320
      });
      d.add(panel);

      var addNode=function(parent,title){        ❷ Adds generic node
        var node=new QxTreeFolder(title);
        if (parent){
          parent.addToFolder(node);
        }
        return node;
      };

      var addLeafNode=function(         ❸ Adds leaf node
        parent,title,image,hasThumbnail
      ){
        var iconUrl=(hasThumbnail) ?
          image+'16.jpg' : null;
        var mainUrl=(hasThumbnail) ?
          image+'.jpg' : image;
        var leaf=new QxTreeFile(title,iconUrl);
        leaf.addEventListener(
          "click",                   ❹ Adds event handler
          function(e){
            panel.set({ source:mainUrl });
          }
        );
        parent.addToFolder(leaf);
        return leaf;
      };

      var tree = new QxTree('Chaos');        ❺ Creates root of tree
      tree.useTreeLines=true;
      tree.set({
        left: 20, top: 48,
        width: 180, height: 320
      });

      var nodeIFS=addNode(tree,'IFS');        ❻ Begins to add child nodes
      addLeafNode(
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    nodeIFS,'Dreamsilk',
    '../assets/images/chaos/ifs/dreamsilk',
    true
  );

  addLeafNode(nodeIFS,'Leaf',
    '../assets/images/chaos/ifs/leaf',true
  );
  addLeafNode(
    nodeIFS,'Spirochete',
    '../assets/images/chaos/ifs/spirochete',
    true
  );
  addLeafNode(nodeIFS,'Trilobite',
    '../assets/images/chaos/ifs/trilobite',true
  );

  var nodeLS=addNode(tree,'L-systems');
  addLeafNode(nodeLS,'Bush',
    '../assets/images/chaos/ls/bush.jpg'
  );
  addLeafNode(nodeLS,'Weed',
    '../assets/images/chaos/ls/weed.jpg'
  );

  var nodeSA=addNode(tree,'Strange Attractors');
  var nodeSA_2D=addNode(nodeSA,'2D');
  nodeSA.addToFolder(nodeSA_2D);
  addLeafNode(nodeSA_2D,'deJong',
    '../assets/images/chaos/sa/sa2d/deJong.jpg'
  );
  addLeafNode(nodeSA_2D,'Lorenz',
    '../assets/images/chaos/sa/sa2d/lorenzII.jpg'
  );
  addLeafNode(nodeSA_2D,'Poly I',
    '../assets/images/chaos/sa/sa2d/quad.jpg'
  );
  addLeafNode(nodeSA_2D,'Poly II',
    '../assets/images/chaos/sa/sa2d/quad2.jpg'
  );

  var nodeSA_3D=addNode(nodeSA,'3D');
  addLeafNode(nodeSA_3D,'Poly I',
    '../assets/images/chaos/sa/sa3d/KRTY_240.gif'
  );
  addLeafNode(nodeSA_3D,'Poly II',
    '../assets/images/chaos/sa/sa3d/MMDW_240.gif'
  );

  d.add(tree);
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      //-->
    </script>
  </head>
  <body>

  </body>
</html>
```

This listing is slightly longer than previous ones, but much of it is repetition of calls to the helper functions that build up the nodes of the tree. Most of the action takes place in the first half, so let's take a look at what's going on.

The `window.application.main()` method should be a familiar starting point for working with qooxdoo by now. We need to create two widgets for this application: the tree control and the preview pane on the right, which will display the relevant image. Initializing the preview pane is straightforward ❶, and we load it up with an arbitrary image from our selection to start with.

When we build the tree, we'll need to repeat ourselves a lot, so we've defined two helper functions to keep the code as short as possible. The qooxdoo tree distinguishes between nodes that do and don't contain children—that is, non-leaf and leaf nodes—so we've provided separate helper methods for each. `addNode()` provides a mechanism for adding non-leaf nodes, or "folders" in qooxdoo parlance ❷, and `addLeafNode()` provides a way to add leaf nodes or "files" ❸. The events on the non-leaf nodes—the opening and closing of the tree—will be handled by qooxdoo automatically, but we'll need to add the events on the leaf nodes ourselves ❹. When we click on the leaf nodes, we want the corresponding image to be displayed in the preview pane. We've also provided an option for leaf nodes to supply a thumbnail image to be rendered in the tree in lieu of the standard icon. As you can see in figure 7.10, we've provided thumbnails for the first four nodes in the tree, just to demonstrate how customizable this component is.

So, now that we've set up our helpers, we can roll up our sleeves and start to assemble the tree. First, we declare the root node of the tree ❺, and then we begin to exercise our helper functions ❻, until we've accounted for all the images in our collection. And that's it—our tree control is fully operational and linked to the preview pane.

Discussion

The tree control is considerably more complex than tabs or toolbars, and it took a little longer to set up, but that's largely due to the increased complexity of our data model, now that we've added the extra categories. Once we put the helper

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

methods in place, we didn't have to deviate into the realms of supporting low-level UI components, and the majority of our code was a description of our domain model.

In this example, the tree contents were just about small enough to be hand-coded. In a much larger set of categories, we might like to make use of Ajax to fetch subcategories on an as-needed basis, but constraints of space prevents us from exploring that option further here.

This concludes our foray into the world of desktop navigation metaphors. In section 7.2, we looked at traditional web-style approaches to navigation, and we can see a considerable gulf between those examples and the ones presented in this section. We can partly account for that gulf based on the division between casual-use apps, which favor web-style navigation, and line-of-business apps, which favor a more sophisticated desktop-style approach. Nonetheless, the territory between the two is far from barren and contains some interesting possibilities. We'll conclude this chapter by looking at the scope for hybrid models of navigation that combine the best features of the Web and the desktop.

7.4 Between the desktop and the Web

Prior to Ajax, the web application was restricted to applications at the periphery of a user's attention, such as shopping/commerce sites, searches, dictionaries, and lookups. A user might consult these several times during a day, but typically only briefly, and as a complement to some other, more complex task. The complex task itself would be handled by a desktop app or a thick client.

Peripheral apps require simple, obvious controls. As Ajax moves into the territory occupied by thick clients, it can adopt the more complex (and more demanding) navigation techniques of the desktop app, such as those we showcased in section 7.3. However, many developers and designers are reluctant to lose the light touch of the web application, and are seeking to combine the best of both worlds. In this section, we'll look at a couple of examples that sit within this middle ground.

7.4.1 The OpenRico Accordion control

The Accordion control from OpenRico is a useful control for reducing information overload and making effective use of constrained space. Users are only exposed to small pieces of content, but they can still quickly navigate to related items.

You may already have heard of the Accordion control by another name: the Outlook bar. It consists of a single content pane, which shows the currently

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

selected content. There are also several bars, which hold the titles of other pieces of content that are also selectable. You can see this clearly in figure 7.11 in a moment. As you click on the bars to access the information they contain, OpenRico animates the bars to progressively hide the old content and display the new. Pretty slick!

Problem

Your application contains a group of several pieces of discrete content but you want only one shown at a time. These pieces of content could be many things: navigations, forms, and so on.

Solution

OpenRico provides a DHTML widget called an Accordion. This widget incorporates several discrete chunks of content and provides a visually appealing method of revealing only one chunk of content at any given time. This control is simple to use, but requires a little bit of setup in the form of some HTML structure; we'll show you how in listing 7.6. First, take a look at the result in figure 7.11.

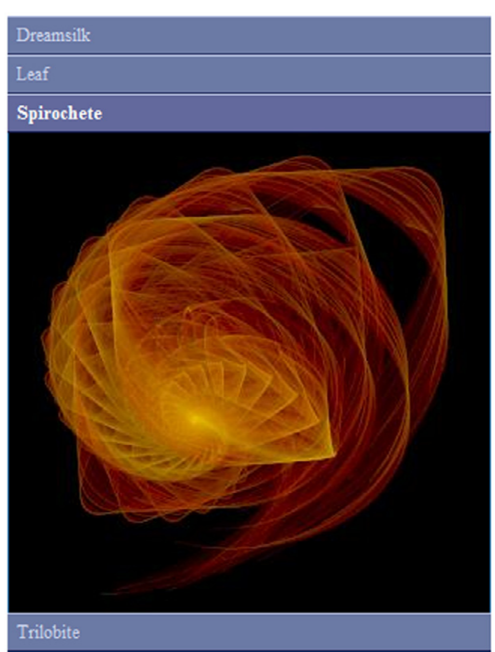

Figure 7.11 OpenRico Accordion control

The first thing you need to do is define a container `<div>` element. In this example we named it `imageAccordionDiv`. Within the container `<div>` we then need to add a subcontainer `<div>` for each chunk of content we're adding to the Accordion. These subcontainers are identified by the suffix `Panel` and will in turn contain two `<div>` elements. The first `<div>` in each set will serve as the title bar, which will be the only visible cue for a content chunk other than the currently displayed one. These title bar `<div>`s can be identified by the suffix `Header`. The second `<div>` in each set will contain the given content chunk. These `<div>`s can be identified by the suffix `Content`.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Once the HTML structure for the Accordion has been defined, we can then pass the JavaScript necessary to initialize the widget. This code appears in the `initialize()` function. The constructor for the Accordion object takes in the container `<div>` element and a properties hash. In this example we are only setting the `panelHeight` property, but many more exist.

One additional note on the code in listing 7.6: in the Accordion constructor, we use the now-familiar `$()` function to retrieve the container `<div>`. We had a look at jQuery's `$()` in early examples, but in this case, we're using the Prototype library, on which Rico is based.

Listing 7.6 Using the OpenRico Accordion widget

```
<html>
  <head>
    <title>Chaotic Images</title>
    <link href='../assets/css/rico.css'
          media='all' rel='Stylesheet' type='text/css' >

    <script src='../assets/js/prototype.js'
type='text/javascript'></script>
    <script src='../assets/js/rico.js'
type='text/javascript'></script>

    <script type='text/javaScript'>
      <!--
window.onload=initialize;

function initialize()
{
  new Rico.Accordion(
    $('imageAccordionDiv'),
    {panelHeight:320}
  );                                    ❶ Constructs Accordion object
}
      //-->
    </script>
  </head>
  <body>
                                                              Declares main container ❷
    <div id='imageAccordionDiv' style='width:322px;overflow:hidden'>
      <div id='dreamsilkPanel'>                  ❸ Declares Accordion panel
        <div id='dreamsilkHeader'
          class='accordionTabTitleBar'>          ❹ Declares per-element title
          Dreamsilk
        </div>
        <div id='dreamsilkContent'>              ❺ Declares per-element content
          <img src='../assets/images/chaos/ifs/dreamsilk.jpg'>
        </div>
      </div>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <div id='leafPanel'>
          <div id='leafHeader' class='accordionTabTitleBar'>
            Leaf
          </div>
          <div id='leafContent'>
            <img src='../assets/images/chaos/ifs/leaf.jpg'>
          </div>
        </div>
        <div id='spirochetePanel'>
          <div id='spirocheteHeader' class='accordionTabTitleBar'>
            Spirochete
          </div>
          <div id='spirocheteContent'>
            <img src='../assets/images/chaos/ifs/spirochete.jpg'>
          </div>
        </div>
        <div id='trilobitePanel'>
          <div id='trilobiteHeader' class='accordionTabTitleBar'>
            Trilobite
          </div>
          <div id='trilobiteContent'>
            <img src='../assets/images/chaos/ifs/trilobite.jpg'>
          </div>
        </div>
      </div>

    </body>
  </html>
```

The `initialize()` function is the real meat of this example. It simply constructs a new Accordion using the contents of the `imageAccordionDiv` `<div>`, and sets the panel height to 320 pixels ❶.

The rest of the code depends on the proper setup of the `<div>`s you wish to be shown in the Accordion. You'll need a container `<div>` for the entire Accordion ❷, which we've labeled `imageAccordionDiv`. Notice that was the name we passed to the Accordion constructor previously ❶. After that, we'll need several subdivision `<div>` elements, which will be displayed inside the Accordion. For each element to be displayed you will need

- A container `<div>` for the Accordion panel ❸
- A title bar `<div>` to display the title, marked up with the `accordionTabTitleBar` CSS class ❹
- A content `<div>` that will display the actual contents ❺

The Accordion object takes care of all the rest!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

The Accordion control is a great way to add some dynamic sparkle to your web applications. It does, however, have its drawbacks. Each content chunk must occupy the same space as all the others. If this is not going to be the case, then the developer will need to apply appropriate CSS styling to allow scrolling within the per-content `<div>` elements.

One interesting thing to note, when comparing the Accordion to the examples in section 7.3, is that OpenRico and qooxdoo take two very different approaches to solving the problem of creating DHTML widgets. In the case of OpenRico, the toolkit requires the developer to lay out the HTML and then takes care of the rest of the work. However, with qooxdoo the aim was to create a complete GUI toolkit, so emphasis was completely on the JavaScript API, and nonexistent on the HTML front, at least where the core framework elements are concerned. You end up with either very little JavaScript and a good bit of HTML footwork, or truckloads of JavaScript and little to no HTML. That choice is up to you.

7.4.2 Building an HTML-friendly tree control

The Rico Accordion has presented an interesting approach to developing interactive navigation controls, in which we declare the elements that compose the widget as plain HTML and then use the JavaScript simply to add the interactivity. This is quite different to qooxdoo's approach, in which the widget is created entirely from JavaScript, and all DOM elements are constructed programmatically. We referred earlier to the interesting territory that lays between the conventional Web and the desktop app approach to navigation and application look and feel, and here we've begun to explore that territory.

A question that naturally arises from this is whether the familiar declarative approach of HTML and CSS can be successfully combined with the interactivity of a pure JavaScript solution, giving us the best of both worlds. Indeed, can we set up the page in such a way that it still functions—albeit less richly—when JavaScript is turned off altogether? In the next example, we're going to look at doing just that for the tree control, which is arguably the most complex and interactive widget that we've looked at so far.

Problem

We have sufficient categories and subcategories within our data to merit the use of a tree control. However, the application must serve a broad audience, and we need to ensure that it is still usable in browsers that have JavaScript turned off.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We don't want to maintain two completely separate codebases for the Ajax and non-Ajax versions of the application, so we need to find a way of accommodating both sets of users within a single design.

Solution

We're facing a pretty tall order! If we understand that the non-JavaScript version of the application won't be as functional as the JavaScript application, then we can make it work.

In the previous example, we noted that the Rico Accordion added behavior to HTML that was declared within the page. That way of doing things will suit our needs here, as we need the unadorned HTML to provide a baseline of functionality. So, what should it look like?

We've chosen a simple interaction model for the HTML application, with each leaf node of the tree a hyperlink directly to the image. After viewing an image, the user can use the back button to return to the tree. That's not a great way of interacting, but it works. Figure 7.12 shows the two stages of interaction with the tree in this mode.

The entire tree contents are shown by default in expanded form and can't be contracted by clicking on the non-leaf nodes. The preview pane isn't used in this

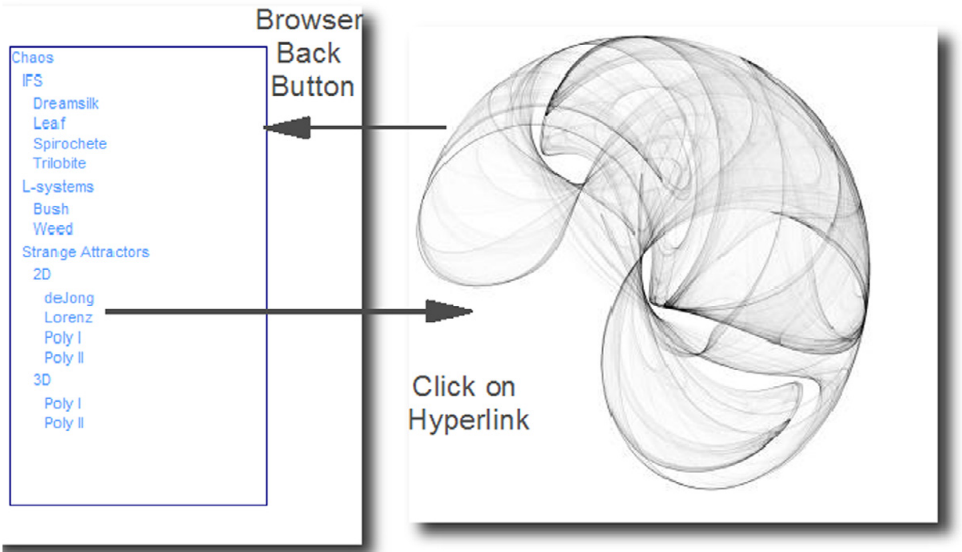

Figure 7.12 Interacting with the tree with JavaScript switched off

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

mode, so it is initially hidden from view. Clicking on a leaf node takes the browser to the full-sized image, from which the user can return to the tree by using the back button. Listing 7.7 shows the HTML for the tree control.

Listing 7.7 HTML for tree control

```
<html>
  <head>
    <title>A Poem Lovely As A Tree</title>
    <link rel="stylesheet" type="text/css" href="main.css">
    <script type='text/javascript'
       src='scripts/prototype.js'></script>
    <script type='text/javascript'
       src='scripts/tree.js'></script>
    <script type='text/javascript'>
      window.onload=function(){
        initTree();
      };
    </script>
  </head>
  <body>
    <div class="pane" id="tree">
      <div class="nodeHeader" id="head_0">Chaos</div>
      <div class="nodeChildren" id="child_0">
        <div class="nodeHeader" id="head_1">IFS</div>
        <div class="nodeChildren" id="child_1">
          <a href='../assets/images/chaos/ifs/dreamsilk.jpg'>
            <div class="nodeHeader leaf"
              id="head_2">Dreamsilk</div>
          </a>
          <a href='../assets/images/chaos/ifs/leaf.jpg'>
            <div class="nodeHeader leaf"
              id="head_3">Leaf</div>
          </a>
          <a href='../assets/images/chaos/ifs/spirochete.jpg'>
            <div class="nodeHeader leaf"
               id="head_4">Spirochete</div>
          </a>
          <a href='../assets/images/chaos/ifs/trilobite.jpg'>
            <div class="nodeHeader leaf"
               id="head_4">Trilobite</div>
          </a>
        </div>
        <div class="nodeHeader" id="head_5">L-systems</div>
        <div class="nodeChildren" id="child_5">
          <a href='../assets/images/chaos/ls/bush.jpg'>
            <div class="nodeHeader leaf" id="head_6">Bush</div>
          </a>
          <a href='../assets/images/chaos/ls/weed.jpg'>
            <div class="nodeHeader leaf" title="ls/weed.jpg"
```

❶ Imports JavaScript

❷ Begins declaring tree nodes

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
                id="head_7">Weed</div>
            </a>
          </div>
          <div class="nodeHeader" id="head_8">Strange Attractors</div>
          <div class="nodeChildren" id="child_8">
            <div class="nodeHeader" id="head_8a">2D</div>
            <div class="nodeChildren" id="child_8a">
              <a href='../assets/images/chaos/sa/sa2d/deJong.jpg'>
                <div class="nodeHeader leaf" id="head_9">deJong</div>
              </a>
              <a href='../assets/images/chaos/sa/sa2d/lorenzII.jpg'>
                <div class="nodeHeader leaf" id="head_10">Lorenz</div>
              </a>
              <a href='../assets/images/chaos/sa/sa2d/quad.jpg'>
                <div class="nodeHeader leaf" id="head_11">Poly I</div>
              </a>
              <a href='../assets/images/chaos/sa/sa2d/quad2.jpg'>
                <div class="nodeHeader leaf" id="head_12">Poly II</div>
              </a>
            </div>
            <div class="nodeHeader" id="head_8b">3D</div>
            <div class="nodeChildren" id="child_8b">
              <a href='../assets/images/chaos/sa/sa3d/KRTY_240.gif'>
                <div class="nodeHeader leaf" id="head_13">Poly I</div>
              </a>
              <a href='../assets/images/chaos/sa/sa3d/MMDW_240.gif'>
                <div class="nodeHeader leaf" id="head_14">Poly II</div>
              </a>
            </div>
          </div>
        </div>
      </div>                                        Declares and hides
                                                          preview pane ❸
      <div class='pane' id='preview' stylle='display:none;'>   <─┘
        <img id='preview_img' src='../assets/images/chaos/ifs/
     spirochete.jpg'></img>
      </div>
    </body>
  </html>
```

The first thing that we do in this page is import the JavaScript code that we'll need ❶. We're going to make use of Prototype here, and we've also moved most of our own code into a separate file, tree.js. In anything but the smallest demo programs, this is a good idea anyway, but in this case, this approach will be specifically useful because we expect some of our users not to be using JavaScript. We don't want to waste their bandwidth by providing too much inline script that they'll ignore.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

As with the qooxdoo tree control (see listing 7.5), there is plenty of repetition here as we assemble the tree, but this time we do it in the HTML ❷. We're making use of the fact that HTML documents have a treelike structure themselves, and the nesting of elements on the page follows the structure of our tree widget. Each node in the tree is composed of a `<div>` element having the `nodeHeader` CSS class, containing the caption for that node. In the case of leaf nodes, these elements have an additional CSS class called `leaf`, and are surrounded with an anchor tag defining the hyperlink. Non-leaf nodes don't have the hyperlink, but do contain a second `<div>` element having the `nodeChildren` CSS class, which is a sibling of the `nodeHeader` element. All child nodes are contained entirely within the `nodeChildren` element, which will allow us to expand and collapse a node when we add the interactivity simply by showing or hiding the child container.

This approach also gives us the basic layout of the tree almost for free, as we add a bit of CSS to ensure there is a visible amount of indentation to the left-hand side of each child container, thereby increasing indentation at each level.

Finally, we declare the preview pane ❸. We'll only want to use this in the JavaScript-enabled version of our app, so we set the style `display` to `none` by default, to hide it from view until we choose to programmatically reveal it.

So, we've catered to the minority of our audience who don't use JavaScript. What will the widget look like for the rest of us? Figure 7.13 shows the results.

Figure 7.13 Tree widget with JavaScript switched on

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

With JavaScript enabled, the tree will be initially shown in the contracted state, with only the root node and the preview pane visible. In the screen shot, we've opened a few selected nodes and clicked on a leaf node, which now opens up the relevant image in the preview pane. How do we do this? Listing 7.8 shows the contents of the tree.js file that we used to rewrite the rules for interactivity.

Listing 7.8 tree.js

```
function initTree(){
  $('preview').show();                          ❶ Shows preview pane
  var allNodes=$$('.nodeHeader');
  var partitioned=allNodes.partition(           ❷ Separates leaf and non-leaf nodes
    function(node){
      return node.hasClassName('leaf');
    }
  );
  var leafNodes=partitioned[0];
  leafNodes.each(
    function(node){
      var anchor=node.parentNode;
      var imgsrc=anchor.href;
      anchor.href='#';                          ❸ Replaces hyperlink with onclick
      node.onclick=function(){
        $('preview_img').src=imgsrc;
      };
    }
  );
  var nonLeafNodes=partitioned[1];
  nonLeafNodes.each(
    function(node){
      var childDivId=node.id
        .replace(/head/,"child");
      var childDiv=$(childDivId);
      node.onclick=function(){                  ❹ Adds event handler
        childDiv.toggle();
      }
      childDiv.hide();
    }
  );
}
```

We can turn the HTML into an interactive tree with surprisingly little code. As with the Rico Accordion, everything is there for us already, and all we need to add is the interactivity. Using a library such as Prototype certainly helps to keep the code brief, too!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Our first task is simple. The preview pane has been hidden, so we make it visible again ❶. The `show()` method is a Prototype extension to the DOM element class. We then use some of Prototype's array functions to assemble all the HTML elements representing tree nodes, and divide them into leaf and non-leaf nodes ❷. We can already identify these elements by their CSS styles, and Prototype supports searching for elements using CSS selectors with the `$$()` function. Having obtained all the node elements, we use the `partition()` method that Prototype has kindly added to the Array class for us, which will return an Array containing two elements. Both elements are themselves Arrays. The first contains all the elements that passed a specified test, and the second all those that failed it. The test is defined as a function object that we pass in as an argument. Our test function simply checks the CSS classes for the node again, to determine whether or not it's a leaf node.

We can then iterate through all the leaf nodes, and fix them up for use in our JavaScript-enabled tree. The first thing we need to do is to deactivate the hyperlink that we had added for the benefit of non-JavaScript users. Having done that, we add a simple programmatic event handler in its place ❸.

Finally, we iterate through the non-leaf nodes. Under the HTML version, these have no interactivity, so we simply need to add it in here, identifying the container element that holds all the children for this node, and toggling its visibility when the title is clicked ❹.

Discussion

The tree control that we've presented here doesn't look quite as sophisticated as the qooxdoo widget from section 7.3.3, but it's not a bad start for a few hours' work. And, depending on your tastes, and the style of your application, the flatter, more web-style UI may be a better fit than qooxdoo's decidedly desktop-like styling. Certainly, there's room for both approaches.

The technique that we've practiced here—allowing a web app to continue to offer functionality as JavaScript is removed—is often referred to as *graceful degradation*. If you need to support a wide range of users, it can be a winning approach, and doesn't entail that much extra effort. By enforcing a clear separation between content and behavior from the outset, we've been able to add the full interactivity with relatively little code.

That concludes our review of navigation techniques and widgets, as well as this chapter. We'll continue to look at the user's workflow in the next chapter, when we examine ways of making Ajax play nicely with a browser's history mechanisms.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

7.5 Summary

When you're thinking of ways to navigate content in an Ajax application, the sky is literally the limit. A large amount of widgets already exist, free for the picking, such as those from qooxdoo and OpenRico.

We're in an interesting phase in the development of thinking about web navigation, with an influx of ideas coming from both the traditional web design world, as we saw in section 7.2, and from the desktop application and thick-client arena, as we saw in section 7.3. These two approaches are beginning to interact with one another, thanks to the disruptive nature of Ajax, which has brought line-of-business applications within the reach of the web application.

In section 7.4, we looked at ways of combining the current thinking from both of these areas. Along the way, we touched on issues such as the separation of design and content from workflow logic, and ways of working with users who can't or won't make use of JavaScript.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

8 Handling back, refresh, and undo

This chapter covers

- Disabling browser navigation features
- The Really Simple History framework
- Handling undo operations

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

One of the greatest problems when designing dynamic content for the Web is the ability for the end user to refresh a page and navigate the browser's history at will. Such tools are great when you're navigating static content, but they open up the proverbial can of worms when using a dynamic web application. For example, with a simple click of the back button or a press of the F5 key, the client-server state becomes out of sync, and if your application makes use of advanced Dynamic HTML (DHTML) techniques like draggable content, the client's layout state is destroyed. Ajax-enabled single-page applications compound these problems even further.

In this chapter we explore a few tricks to prevent the end user's access to history navigation and page refreshing. These tricks include opening a new browser window with all toolbars removed, disabling any keyboard shortcuts used for navigating history, and disabling the right-click context menu. We'll also look at some techniques for working with these browser features, such as using hashes to store application state in the URL, using the Really Simple History framework to easily add bookmarking and history navigation functionality to a single-page Ajax application, and implementing your own undo stack.

8.1 Removing access to the browser's navigation controls

Removing access to the browser's navigation controls is a threefold procedure. We must deny the user access to the various toolbars that contain navigation functionality; we must trap any keyboard shortcuts that allow navigation; and we must disable the context menu. It is important to keep in mind that end users may not be thrilled with this forceful narrowing of their user experience, which means you'll have to provide a way for them to easily navigate your application. Let's take a look at how best to handle these issues.

8.1.1 Removing the toolbars

To remove the address and navigation bars, a new window must be opened programmatically using JavaScript. It is not possible to add or remove toolbars from an existing browser window. Because of this limitation, you'll have to create a launchpad page from which you can spawn a new window containing your application. The JavaScript API for opening a new window is fairly simple and straightforward:

```
window.open(URL, name, options, replace);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The `window.open()` method provides you with a wide range of customization options for the window to be opened. Tables 8.1 and 8.2 provide an in-depth look at the features available for customization.

Table 8.1 window.open() parameters

Parameter	Type	Description
`URL`	String	Specifies the location of the page you wish to display. An empty string may be passed if you do not wish to initially load a page (this is helpful if you wish to dynamically generate content for the window via scripting).
`name`	String	Specifies the `name` property of your new window. The name of a window allows the window to be referenced using the same constructs as a frame within a frameset. For example, a hyperlink of the form `<a target='thewindow' href='thepage.html'>` will display `thepage.html` in the window with the name `thewindow`. If the name refers to a window that already exists, then `window.open()` will display the content in that window, instead of opening a new one.
`options`	String	Optional. Specifies the options available to the new window. This parameter may contain one or more `key=value` pairs separated by commas. Valid values for Boolean options are `yes`, `no`, `1`, or `0`. You may leave off any of the Boolean options if you wish them to default to false.
`replace`	Boolean	Optional. If true, the new location will replace the current one in the browser's history. This parameter may not be supported on some browsers.

Table 8.2 window.open() commonly supported options

Option	Type	Description	Default
`width`	Integer	The width in pixels of the window	Same as parent
`height`	Integer	The height in pixels of the window	Same as parent
`left`	Integer	The x-coordinate of the top-left corner of the window	Auto
`top`	Integer	The y-coordinate of the top-left corner of the window	Auto
`scrollbars`	Boolean (`yes`, `no`, `1`, or `0`)	Determines if scrollbars are available	Yes

continued on next page

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Table 8.2 window.open() commonly supported options *(continued)*

Option	Type	Description	Default
resizable	Boolean (yes, no, 1, or 0)	Determines if the window is resizable	Yes
toolbar	Boolean (yes, no, 1, or 0)	Determines if the toolbar should be displayed	Yes
location	Boolean (yes, no, 1, or 0)	Determines if the address bar should be displayed	Yes
directories	Boolean (yes, no, 1, or 0)	Determines if the links bar should be displayed	Yes
status	Boolean (yes, no, 1, or 0)	Determines if the status bar should be displayed	Yes
menubar	Boolean (yes, no, 1, or 0)	Determines if the menu bar should be displayed	Yes

A common use of this function is to open a window without any toolbars, in essence an undecorated window. A generic function for opening an undecorated window might look like this:

```
function openWithoutToolbars(URL, windowName) {
    window.open(
        URL,
        windowName,
        'status=1,scrollbars=1,resizable=1',
        true
    );
}
```

So, we've removed the visible buttons, but the user can still exercise the same functionality by using the keyboard or the context menu. Let's see how to remove that access as well.

8.1.2 *Capturing keyboard shortcuts*

Capturing keyboard shortcuts involves adding an event handler at the document level to intercept the appropriate keyboard shortcuts. (We discussed the JavaScript event model in chapter 5.) There are eight common keyboard shortcuts for controlling navigation and the state of the currently loaded page, as table 8.3 shows.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Table 8.3 Keyboard shortcuts for navigating history

Shortcut	Description
Backspace	Navigate backward in history
Alt/Option+left arrow	Navigate backward in history
Alt/Option+right arrow	Navigate forward in history
Ctrl/Command+left arrow	Navigate backward in history
Ctrl/Command+right arrow	Navigate forward in history
F5	Refresh window
Ctrl/Command+R	Refresh window
Ctrl/Command+H	Show history
Alt/Option+Home	Go to home page

To detect these keypresses, we must attach a `keydown` event (and in the case of Mozilla/Firefox we must also attach a `keypress` event) at the document level to check which key was pressed as well as any relevant modifiers to detect the key combinations and prevent the event from propagating further. Also, since the backspace key is one of the eight shortcuts, we need to add a special case to allow that key to be processed in the event that the end user is pressing the backspace key while a text area or input element has input focus.

We'll present an example of this technique shortly, in section 8.1.4. For now, let's move on to the next feature: the context menu.

8.1.3 Disabling the right-click context menu

The *context* menu is the detached menu that appears when a user right-clicks the mouse on the browser window's content area. The context menu contains some navigation features, so we need to disable this menu as well. Most newer browser versions provide an event called `oncontextmenu` that fires when an end user right-clicks the mouse. To disable this context menu, simply register an event at the document level for the `oncontextmenu` event that prevents the event from propagating.

OK, that's everything in our checklist covered. Let's put our new knowledge into practice and look at a working example.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

8.1.4 Preventing users from navigating history or refreshing

When developing Ajax applications, it sometimes becomes necessary to prevent users from navigating the browser history and from refreshing the page. Because the UI is so dynamic, a page refresh or history navigation will destroy the state of the current application, and will cause the user to lose their work instead of taking them back to the previous page.

Problem

You have developed a single-page Ajax application and you need to remove from the user the ability to navigate the browser's history or refresh the page.

Solution

The first step in creating a browser window with all functionality removed is to provide a mechanism to open a new window with all of the toolbars removed. You can see the results of this in figure 8.1. We'll create a launchpad (listing 8.1) to serve as the starting point for a registration page; this page will not allow the end user to navigate through history, browse to another page, or refresh/reload the contents of the registration page.

Listing 8.1 Launchpad page

```
<html>
  <head>
    <title>Application Launchpad</title>
    <script type='text/javascript'>

function openWithoutToolbars(URL, windowName, width, height) {
    window.open(
        URL,
        windowName,
        'status=1,scrollbars=1,resizable=1'+      <-- A Values for removing toolbars
        (width  ? ',width='+width   : '')+
        (height ? ',height='+height : ''),
        true
    );
}

function openRegistration() {
    openWithoutToolbars(
        './registration.html',
        'REGISTRATION',
        480, 580
    );
}

    </script>
  </head>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    <body>
      <a href='javascript:openRegistration();'>
        Launch Registration
      </a>
    </body>
  </html>
```

Figure 8.1 Registration app without toolbars

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The second step (shown in listing 8.2) is to disable all of the available keyboard shortcuts for history navigation and page refreshing from the newly opened window. In keeping with our maxim of using tried-and-tested third-party libraries, we'll use the Prototype library's cross-browser event-registration mechanism here, so we'll need to reference `prototype.js` in our page. The two key methods that Prototype provides for us here are `Event.observe()`, which registers an event on an object, and `Event.stop()`, which prevents the event from propagating.

Listing 8.2 Launchpad JavaScript

```
var isHistoryShortcutDisabled = false;

Event.observe(                          <-- 1 Disables context menu
  document,
  'contextmenu',
  function(event){
    Event.stop(event);
    return false;
  }
);
Event.observe(document,'keypress',
  checkHistoryShortcutDisabled);          2 Disables hotkeys
Event.observe(document,'keydown',
  disableHistoryShortcuts);

function disableHistoryShortcuts(event) {
    var targetTag = Event.element(event).tagName;
    var isTextInput = (
      (targetTag == 'TEXTAREA')
      || (targetTag == 'INPUT')
    );

    var keyCode = event.which || event.keyCode;
                                                     3 Suppresses
    if (( keyCode == 116) ||                    <--    navigation keys
        ((keyCode ==   8) && (!isTextInput))    ||
        ((keyCode ==  36) && event.altKey)      ||
        ((keyCode ==  37) && event.altKey)  ||
        ((keyCode ==  39) && event.altKey)  ||
        ((keyCode ==  37) && event.ctrlKey) ||
        ((keyCode ==  39) && event.ctrlKey) ||
        ((keyCode ==  82) && event.ctrlKey) ||
        ((keyCode ==  72) && event.ctrlKey)) {
        isHistoryShortcutDisabled = true;
        Event.stop(event);
        return false;
    }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function checkHistoryShortcutDisabled(event){      <-- 4 Disables Mozilla keypress
    if (isHistoryShortcutDisabled) {
        isHistoryShortcutDisabled=false;
        Event.stop(event);
        return false;
    }
}
```

First, we disable the context menu ❶. All we want to do in the callback is stop the event from propagating, so we define the function inline. Trapping the keypresses ❷ requires a bit more thought, so we've defined the callback functions separately.

The main callback function is `disableHistoryShortcuts()`, which is registered against the `keydown` event. In this function, we need to identify the navigation hotkeys that we'll suppress ❸ and prevent propagation only in those cases. This requires us to, among other things, figure out if we're inside a text input field or, indeed, any other input field that responds to key presses, such as a dropdown list. We also need to trap the `keypress` event in Mozilla ❹. We register the callback in any case—under Internet Explorer, the registration will have no harmful side effects. Finally, as a result of our efforts, we'll have a window that can be neither navigated nor refreshed.

Discussion

We have just seen a complete solution for removing the end user's ability to navigate history or refresh a page. This technique, even though it is quite effective, may make your users unhappy with your application. None of us like to have our freedoms revoked, and for some end users it will feel as if you have done just that: deprived them of their ability to view and navigate your web application the way they are accustomed to. If you decide to employ this technique, please keep that in mind.

We need to empower users again, having hijacked their browser-given rights of willy-nilly navigation. It is extremely important to provide alternative methods of navigation if you do override the native browser navigation controls. If you don't give your users alternative methods, or even ways of bookmarking, they won't be likely to use your application. The examples of Google's GMail and Maps applications spring to mind. Even though they have a dynamic, client-side interface, they still allow the user to navigate backward and forward through the application (and even make bookmarks) using the native browser controls. They are just subverting those controls to their own purposes. Next up, we'll show you exactly how to do that.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

8.2 Working with a browser's navigation controls

If you wish to provide a richer and less restrictive user experience (both of these will make the end user much happier), you have to work with the browser's navigation and refresh features. This can be a daunting task. How do you maintain state if the user refreshes their window, clicks the back button, or goes to a completely different site and then navigates back to your application? There are several techniques you can use to hold the state of your single-page application between refreshes and even to provide logical bookmarks so that when the user clicks the back or forward button they aren't in for a nasty surprise (the application resetting to some default state or even worse). Instead, the user can step backward through their actions.

8.2.1 Using the JavaScript history object

With the *history object*, JavaScript provides a way to programmatically navigate the browser's history. Using this object, you can emulate the browser's back and forward buttons, provide a link in a dynamic web app to take you back to the previous page (even if the current page has multiple points of entry), or even force the browser to always show the last page in history by adding

```
window.onload = function() {history.go(1);}
```

to all of your application's pages. However, this is a pretty inelegant hack that will likely get you condemned by the web development community. The properties and functions of the history object appear in tables 8.4 and 8.5, respectively.

Table 8.4 Property of the history object

Property	Description
`length`	The number of entries in the history object

Table 8.5 Functions of the history object

Function	Description
`back`	Loads the previous URL in the history list.
`forward`	Loads the next URL in the history list.

continued on next page

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Table 8.5 Functions of the history object *(continued)*

Function	Description
`go`	Goes to a specific URL in the history list. The parameter *where* can be an integer or a string. In the event of an integer, goes to the URL with the specific position relative to the current document. For example, -1 goes back one page, and 1 goes forward one page. In the event of a string, goes to the first URL that matches the string, either completely or partially.

8.2.2 Hashes as bookmarks

Hashes are those bits of a URL that hang out at the end and are prepended with a # symbol. In typical website use, hashes indicate that the browser should focus on a named anchor tag. Note that the only way to update the location of the browser without causing a reload of the page in its entirety is through the use of a hash. You can access this value by using a smattering of regular expressions:

```
var hash = window.location.href.replace(/(.*)(#)(.*)/,'$3');
```

The hash value can be used for all sorts of things, but it's best used for capturing a snapshot of the application state. You can use it to directly store small amounts of application state, or as a key to a larger, more complex state snapshot. For example, if you have a weather service application that shows the weather for many locations, you can allow end users to bookmark a page after they have already selected the location they wish to view. That way, on their next visit they don't have to reselect the location.

Problem

You need to provide a reference point in your application that the end user can bookmark for later viewing.

Solution

We'll start off with a very simple example here, in which the JavaScript code will watch the URL of the current page for changes in the hash value and update the contents of a form accordingly. The rather minimalistic UI for this example is shown in figure 8.2. The code required to implement this example appears in listing 8.3.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

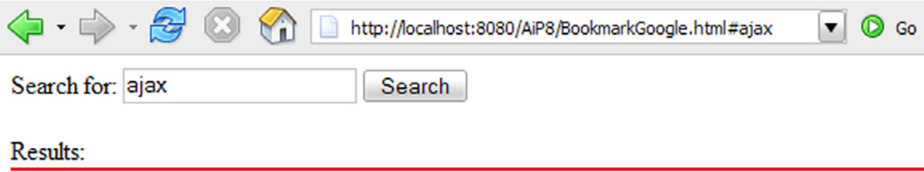

Figure 8.2 Simple bookmarking example. The contents of the text input box will be set to match the hash at the end of the URL.

Listing 8.3 Hashing JavaScript

```
var ajaxRequest;
var currentHash;

setInterval('checkHash()', 250);                    <-- 1 Sets checking interval

function checkHash() {
    var newHash = getHash();
                                                        2 Checks current
    if (newHash && (newHash != currentHash)) {      <--   view status
        currentHash = newHash;
        getBookmark();                <-- 3 Makes server request
    }
}
                                      4 Extracts hash
function getHash() {              <--
    if (window.location.href.indexOf('#') > -1) {
        return window.location.href.replace(/(.*)(#)(.*)/,'$3');
    } else {
        return null;
    }
}

function getBookmark() {              5 Creates Ajax request
    new Ajax.Request(             <--
      '/servlet/Bookmarks?bookmarkId='+currentHash,
      { method: 'get',
        onComplete: function(xhr){          <--
          eval(xhr.responseText);               6 Evaluates response
        }
      }
    );
}
```

The code in listing 8.3 checks for a hash in the `window.location.href` property ❹, and if one exists we'll also verify that it doesn't equal the bookmark being

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

currently viewed ❷. This function, called `checkHash()`, will be executed every quarter-second ❶. If a new hash value is found, `checkHash()` ❸ will call `getBookmark()`, which will use an XMLHttpRequest object (wrapped up tidily here in Prototype's Ajax.Request) to talk to the server ❺, passing the hash as the bookmark ID. The callback will then evaluate the JavaScript returned by the server to render the bookmark state ❻.

Discussion

This was a simple example of what can be accomplished using this powerful technique. It is very much like passing values via the query string, but with the added benefit of being able to use Ajax to update only a portion of the current document instead of requiring a full page reload.

With state-based hash codes, you can now give users a way to bookmark their application in the state it was in. You may be interested in chapter 11, which discusses client-side state management. You should be able to store the state of the application on the client side instead of the server side. This state is indexed with a hash, as we've just discussed. When a user goes to the bookmarked hash, the state can then be retrieved from the client's state cache instead of the server. You won't need to worry about maintaining state on your already-stressed servers; simply push the responsibility to the client.

Now that we've allowed our users to maintain bookmarks to dynamic user interfaces, let's move on to another pain point: maintaining dynamic interface history. The Really Simple History framework is here to show us the way and uses the technique of URL hashing we've just discussed.

8.2.3 Introducing the Really Simple History (RSH) framework

Continuing our exploration of browser history and state, we see that there are a multitude of frameworks for working with history in Ajax applications. The one that stands out as far as ease of use and its ability to be standalone is Brad Neuberg's Really Simple History (RSH) framework, available at http://codinginparadise.org. RSH provides you with the capability to store a series of history events as the end user interacts with your application. Each event is associated with a hash in the document's URL. When the user clicks the browser's back and forward buttons, RSH uses the hash values to retrieve the event associated with it and calls any listeners that are registered with it to process the event.

RSH provides a great deal of functionality, but for now we'll focus on the basics. The mechanism RSH uses for managing history state is the dhtmlHistory

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

object. The dhtmlHistory object provides four major methods (table 8.6) for initializing itself, registering listeners, getting the current location, and adding history events.

Table 8.6 Functions of the dhtmlHistory object

Function	Description
`initialize`	Initializes the dhtmlHistory object. Should be called in `window.onload`.
`getCurrentLocation`	Returns the location String for the current page.
`addListener`	Adds a history event listener for handling history change events.
`add`	Stores a history event. When the page's URL changes and contains the value of location as its hash, the history event listener is called and passed the location and data of the event. `add()` takes two parameters. The first is the key to the event, which will show as a hash in the URL. The second is the data object associated with the event.

Let's see some examples of RSH in action, shall we?

8.2.4 Using RSH to maintain state at the client level

In this example and the next, we'll use RSH to enhance the tree widget from section 7.4.2. While using the tree, the state of the page changed several times, but none of these changes would be captured by the browser history system. This could be disconcerting for our users. In the next example, we'll use RSH to implement a client-side state management system that plays nicely with the back and forward buttons.

Problem

Your application contains programmatic transitions that your user might think of as "new pages." They will therefore expect the history buttons to work with these changes, and may be frustrated if they don't. You need to provide state management to your single-page application while the user is using it. You only need to maintain the state during a client session, and can therefore code the state management on the client only.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Solution

This example will use the HTML-based tree that we wrote in chapter 7 to serve as the treeview component, as shown in figure 8.3.

RSH is a flexible framework, as you'll see, and our first task is to decide which changes of application state are significant enough that we want to record them in the browser history. We've decided to include both the expand/collapse of a node, as well as the change in the preview image when a leaf node is clicked. We'll therefore need to add a history entry when either of these events occur, and store sufficient information with it to restore the application state when the history is recalled via the back or forward button.

The code that we wrote for the tree in chapter 7 already contains action handlers for expanding/collapsing nodes, and for changing the preview image. We'll define a new helper function in this example that records the current state of the history when either of these events occurs, using the `dhtmlHistory.add()` method. We'll also register a listener with the dhtmlHistory object that will restore the tree state to a previously recorded value. Listing 8.4 contains the full JavaScript listing for our history-aware tree. Additional code not present in the previous version (see listing 7.8) appears in bold.

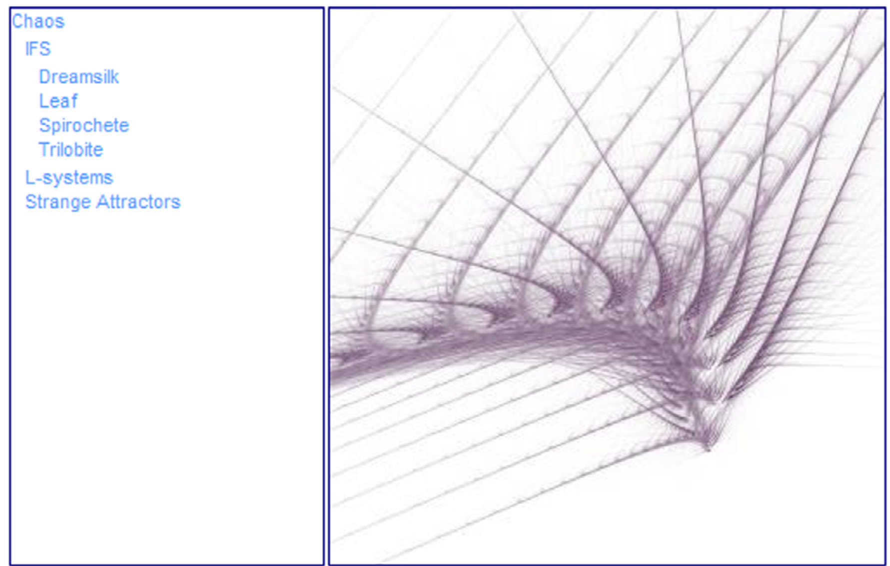

Figure 8.3 The tree component from chapter 7. In this chapter, we'll add support for the back button.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 8.4 tree.js with added history

```
var historyCount=0;

function initTree(){
var leafNodes=null;
var nonLeafNodes=null;

  $('preview').show();
  var allNodes=$$('.nodeHeader');
  var partitioned=allNodes.partition(
    function(node){
      return node.hasClassName('leaf');
    }
  );
  leafNodes=partitioned[0];
  leafNodes.each(
    function(node){
      var anchor=node.parentNode;
      var anchorParent=anchor.parentNode;
      var imgsrc=anchor.href;
      anchor.removeChild(node);
      anchorParent.replaceChild(node,anchor);
      node.onclick=function(){
        $('preview_img').src=imgsrc;
        addHistory();            <-- 1 Registers change of preview
      }
    }
  );
  nonLeafNodes=partitioned[1];
  nonLeafNodes.each(
    function(node){
      var childDivId=node.id.replace(/head/,"child");
      var childDiv=$(childDivId);
      node.onclick=function(){
        childDiv.toggle();
        addHistory();            <-- 2 Registers expand/collapse of node
      }
      childDiv.hide();
    }
  );

  dhtmlHistory.initialize();     <-- 3 Initializes history
  dhtmlHistory.addListener(      <-- 4 Adds listener function
    function(location,data){
      if (data){
        setTreeState(data);
      }
    }
  );
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    if (!dhtmlHistory.getCurrentLocation()){     ⑤ Records initial
      addHistory();                                 history state
    }
  }

  function addHistory(){            ⑥ Records history entry
    dhtmlHistory.add(
      'history'+historyCount,
      getTreeState()
    );
    historyCount++;
  }

  function getTreeState(){
    var treeState={
      nodes:{},
      image:$('preview_img').src
    };
    nonLeafNodes.each(
      function(node){
        var childDivId=node.id.replace(/head/,"child");
        var childDiv=$(childDivId);
        var isOpen=childDiv.visible();
        treeState.nodes[childDivId]=isOpen;
      }
    );
    return treeState;
  }

  function setTreeState(state){
    for (node in state.nodes){
      var nodeDiv=$(node);
      if (nodeDiv){
        if (state.nodes[node]){
          nodeDiv.show();
        }else{
          nodeDiv.hide();
        }
      }
    }
    $('preview_img').src=state.image;
  }
```

The changes to our application are dictated by the lifecycle of the dhtmlHistory object, as outlined in table 8.6. First, we need to initialize the history object ❸. Immediately after doing that, we register a listener function ❹, which will be called for us by the dhtmlHistory object when the user uses the back or forward button. We then record the initial state as the first history item ❺, and modify

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the event handlers on the tree to add new history elements when we change the preview image ❶ or expand or collapse a node ❷. To keep the code clean, we've provided a single helper function `addHistory()` ❻, which can be used in all three cases.

Each entry in the history needs to be uniquely named. We've adopted a straightforward approach here, simply incrementing a global counter every time we write to the history.

The state of our application consists of the position of each node in the tree (i.e., whether it is opened or closed) and the image showing in the preview. We've provided two helper functions, `getTreeState()` and `setTreeState()`, to read and write this state, respectively. This keeps the interaction with the history object simple and separate from the internal logic of our tree.

Note that in listing 7.8, we switched off the anchor nodes by setting the `href` attribute to a single hash sign, as in

```
anchor.href='#';
```

Here, we can't do that, because RSH is using the hashes in the URLs in a more meaningful way. So, we've had to do a bit more DOM manipulation wizardry to remove the hyperlinks altogether without losing the text inside them. It's a bit like the old stage magician trick of whipping the tablecloth away without disturbing the plates or glasses, but without the danger of creating a mess!

Discussion

The image browser application should now be capable of maintaining its state. Try expanding and collapsing multiple nodes and view some of the images. Now use the back and forward buttons to navigate through the history of your actions.

Users will be pleased that you have returned to them the simple semantics of the back and forward buttons. The RSH framework makes this type of behavior easy to implement by just maintaining state through URL hashes along with some secondary information associated with each hash. RSH allows us to make this information as simple or as complex as the application requires.

This example showed us how to maintain state on the client side, which will persist as long as a single session. In some cases, we may wish to preserve application state between sessions or across machines. In this case, we'll need to store the session data on the server. We'll do that in the next example, and see what changes we have to make to our approach.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

8.2.5 Using RSH to maintain state at the server level

Now that you know how to use RSH to maintain state on the client side, let's take a look at how to use it to maintain state at the server side.

Problem

You need to preserve history between user sessions, and thus need to store the client state on the server.

Solution

To demonstrate the use of RSH to maintain state at the server level, we'll ask RSH to store just the location/keys of the events and move the persistence of the actual state to the server, where it will be stored in the session. (If we wanted a more robust solution, we could persist the state to a file or database, but we want to keep the server examples simple here.) For this, we need Ajax. When we store a history state, we'll send the client state to the server, along with the key used to identify it on the client. Similarly, when a history event is triggered by the back button, an Ajax request containing the location/key will be sent to the server, which will respond with a block of data describing the historical state of the client.

First, let's take a look at the changes required on the client. Listing 8.5 provides the details, with changes from listing 8.4 shown in bold.

Listing 8.5 tree.js with server-side history maintenance

```
var leafNodes=null;
var nonLeafNodes=null;
var historyCount=0;

function initTree(){

  $('preview').show();
  var allNodes=$$('.nodeHeader');
  var partitioned=allNodes.partition(
    function(node){
      return node.hasClassName('leaf');
    }
  );
  leafNodes=partitioned[0];
  leafNodes.each(
    function(node){
      var anchor=node.parentNode;
      var anchorParent=anchor.parentNode;
      var imgsrc=anchor.href;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        anchor.removeChild(node);
        anchorParent.replaceChild(node,anchor);
        node.onclick=function(){
          $('preview_img').src=imgsrc;
          addHistory();
        }
      }
    );
    nonLeafNodes=partitioned[1];
    nonLeafNodes.each(
      function(node){
        var childDivId=node.id.replace(/head/,"child");
        var childDiv=$(childDivId);
        node.onclick=function(){
          childDiv.toggle();
          addHistory();
        }
        childDiv.hide();
      }
    );

    dhtmlHistory.initialize();
    dhtmlHistory.addListener(
      function(location,key){
        if (key){
          fetchTreeState(key);
        }
      }
    );
    if (!dhtmlHistory.getCurrentLocation()){
      addHistory();
    }
  }

  function addHistory(){
    var data=getTreeState();
    var key='history'+historyCount;
    historyCount++;
    new Ajax.Request(                        <-- 1 Sends Ajax request
      "jsp/treeState.jsp",
      {
        method: "post",
        parameters: $H({
          key:key,                           <-- 2 Encodes state as JSON
          data:JSON.stringify(data)
        }).toQueryString(),
        onComplete:function(response){
          var responseObj=                   <-- 3 Parses JSON response
            JSON.parse(response.responseText);
          if (responseObj &&
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          responseObj.status=="ok"){
          dhtmlHistory.add(key,key);
        }
      }
    }
  );
}

function getTreeState(){
  var treeState={
    nodes:{},
    image:$('preview_img').src
  };
  nonLeafNodes.each(
    function(node){
      var childDivId=node.id.replace(/head/,"child");
      var childDiv=$(childDivId);
      var isOpen=childDiv.visible();
      treeState.nodes[childDivId]=isOpen;
    }
  );
  return treeState;
}

function fetchTreeState(key){          <-- 4 Gets historical state from server
  new Ajax.Request(
    "jsp/treeState.jsp",
    {
      method:"get",
      parameters: $H({ key:key }).toQueryString(),
      onComplete:function(response){
        var responseObj=
          JSON.parse(response.responseText);
        if (responseObj){
          updateTreeState(responseObj);
        }
      }
    }
  );
}

function updateTreeState(state){       <-- 5 Updates client state
  for (node in state.nodes){
    var nodeDiv=$(node);
    if (nodeDiv){
      if (state.nodes[node]){
        nodeDiv.show();
      }else{
        nodeDiv.hide();
      }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    }
  }
  $('preview_img').src=state.image;
}
```

We make use of Ajax (using Prototype's Ajax.Request) to record each entry in the history ❶. In the previous example, we recorded the state as JavaScript object literals, so it seems a natural choice to use JSON to encode the data when sending it to the server ❷. We're making use of the same json.js library that we discussed in chapter 2. The response is returned as JSON, too ❸. Note that we also need to store the history on the client, using the `addHistory()` helper function that we introduced in the previous example. We could do so at the same time that we send the request, but we've chosen to defer that action until the response has come back from the server telling us that the history has been stored. That way, we can be sure that the data is available for us when we invoke the history.

When we reinstate a historical state of the application, in response to the back button, we need to contact the server again ❹. Having fetched the state from the server, we can then parse the JSON data it contains and update the client as before ❺. (We've changed the name of `setTreeState()` to `updateTreeState()` to better reflect its role, but the code remains the same.)

So, we're sending data back and forth to the server, but what do we do with it when it gets there? Listing 8.6 shows the simple JSP that stores our history data in the user session.

Listing 8.6 treeState.jsp

```
<jsp:directive.page
  contentType="text/json"
  import="java.util.*"
/>
<%
String key=request.getParameter("key");
String data=request.getParameter("data");
if (data==null){
  %><%=session.getAttribute(key)%><%
}else{
  session.setAttribute(key,data);
  %>{ "status" : "ok" }<%
}
%>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We promised to keep the server-side code simple, and we've certainly done that! We're not making any use on the server of the history state that we're storing, so we don't bother to decode the JSON strings that we receive but simply store them as strings in the session. As noted at the outset of this example, we've chosen the session for its simplicity. If we wanted the user to be able to retrieve their history across sessions, or machines even, we'd need to use a file or a database for more persistent storage.

Discussion

This example, when run, should exhibit the same behavior as the previous serverless example except that the state's data persistence and rendering logic is now handled at the server level. The possible uses for this technique are endless and provide you with another great tool.

Server-side state persistence has a few advantages, as well as disadvantages, compared to client-side state persistence. One of the advantages is that users are now no longer bound to their local browser. They can log on from any client and resume their session as they left it, with the full history of their previous session available to them. If there is an error in the application, or it behaves in an unexpected way, the labor of troubleshooting the application is now reduced. Developers can look through a user's history to pinpoint where in the history the application diverged from the expected behavior. User sessions can be recorded for playback later, and extensive analysis of user behavior patterns is now possible.

Of course, one disadvantage is that more code and thinking is required on the server side in order to provide users with this capability. Because we now start creating round-trips to the server, the issues of latency and bandwidth rear their ugly heads; a slow connection to the server can negatively impact the user's experience when they attempt to navigate backward and forward through the application. From a user privacy standpoint, if we start recording so much information about user behavior patterns, we can become quagmired in privacy issues. Extensive care will need to be taken to protect the collected data.

A topic closely related to handling state and history is handling some sort of undo mechanism. We'll explore how to add undo capabilities to an application in the next section.

8.3 Handling undo operations

Browsers provide some undo/redo capability, but this capability is largely limited to editing forms. If you wish to allow users to undo and redo changes to custom,

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

rich DHTML controls, you must implement your own capability. This isn't as hard as it might first appear. A basic undo system needs to do four things:

- Keep track of user actions, placing them in a stack-type data structure as they occur
- Provide methods for traversing the stack, both forward and backward (in this capability, our undo system is not a genuine stack, as actions popped from the stack will remain there in case we want to redo them)
- Provide capability for applying the actions as they are accessed
- Keep an index of the current position in the stack

There are two more important things to remember when implementing an undo stack. First, for each action stored in the stack, you need to keep track of both the before and after states of the action. For example, if an undo action involves a character input into a text field, the value of the text field before the new character was added needs to be stored as well as the value after the new character was added. This way, the stack can be traversed backward (undo) as well as forward (redo). Second, if an action is added somewhere in the middle of the stack, all actions that have a higher index than the added action at the time of insertion are deleted. For example:

1. A user accomplishes 10 undoable actions.
2. The user then undoes 5 of those actions.
3. The user then performs another undoable action.
4. The undo stack now contains 6 actions; the original 1–5 are retained, as well as the new action at position 6. The original 6–10 are no longer valid, and are thus removed.

8.3.1 *When to provide undo capability*

Figuring out when to provide undo capability in an application depends on a great deal of variables, among them the type of application, the expected user base, and machine performance. As a general rule, undo capability should be provided only when an action occurs that changes small, manageable chunks of data. Complex actions that cannot be undone easily should obviously be avoided, along with actions that require server round-trips with large amounts of data passing back and forth.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

8.3.2 Implementing an undo stack

For this example, we'll create an undoStack object as a top-level variable. It will contain an index variable and a stack in the form of an array. We'll provide an event listener to detect when Ctrl+Z and Ctrl+Y keyboard shortcuts are pressed, and also allow undo and redo buttons or other UI mechanisms to operate the stack.

The object that we've described so far handles the generic functionality of maintaining an undo and redo stack. We haven't touched on the issue of what happens when an undo or redo operation occurs, as this will depend on the nature of the application. Taking inspiration from the RSH library from the previous examples, we'll provide a callback mechanism that will allow the user to define how the application state responds to undo and redo. So, let's see what this undo stack looks like.

Problem

You need to provide undo capability for actions that the browser's undo/redo functionality doesn't necessarily support, or you need to replace the browser's undo/redo functionality with a customized version of your own.

Solution

We'll start off with a simple example here by attaching the undo stack to a couple of text input fields. There are two aspects to coding this. First, we need to implement the generic undo stack that we discussed earlier, and second, we need to hook it up to the text inputs. Let's look at the undo stack itself first; listing 8.7 presents the full code.

Listing 8.7 undo.js

```
var undoStack={
    curIdx: 0,
    stack: [],
    undoHandler: null,
    doDiffCheck: true,
    actionPerformed:false,                    ❶ Initializes stack
    init: function(theUndoHandler){      <-
        Event.observe(document, 'keydown',
          this.checkUndo);
        Event.observe(document, 'keypress',
          this.postCheckUndo);
        this.undoHandler=theUndoHandler;
    },                                          ❷ Handles undo keypresses
    checkUndo: function(event){              <-
        var keyCode = event.which || event.keyCode;
        if((keyCode ==  90) && event.ctrlKey){
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            undoStack.undo();
            undoStack.actionPerformed=true;
            Event.stop(event);
            return false;
        }else if((keyCode ==  89) && event.ctrlKey){
            undoStack.redo();
            undoStack.actionPerformed=true;
            Event.stop(event);
            return false;
        }
    },
    postCheckUndo: function(event){
        if(undoStack.actionPerformed){
            undoStack.actionPerformed=false;
            Event.stop(event);
            return false;
        }
    },
    seek: function(index){
        if(index >= 0 &&
          index < this.stack.length){
            this.curIdx=index;
        }
    },
    add: function(theType,theValue,
      ignoreDiffCheck){                          ❸ Adds action to stack
        var action =this
          .getNewUndoAction(theType,theValue);
        var success=false;
        var differs=(
          this.doDiffCheck && !ignoreDiffCheck)?
            (this.checkAction(action)):
            true;
        if(differs){
            this.stack[this.curIdx++]=action;
            this.stack.length=this.curIdx;
            success=true;
        }
        var stateAction=this.getNewUndoAction();
        stateAction.canUndo=true;
        stateAction.canRedo=false;
        this.undoHandler(stateAction);
        return success;
    },
    checkAction: function(action){
        var latest = this.stack[this.stack.length-1];
        return (latest) ?
            (!action.equals(latest)) :
            true;
    },
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    undo: function(){                    ❹ Specifies undo action
        var action=null;

        if(this.curIdx > 0){
            action=this.stack[--this.curIdx];
        }else{
            action=this.getNewUndoAction();
        }
        action.canUndo=this.curIdx > 0;
        action.canRedo=this.curIdx < this.stack.length;
        this.undoHandler(action,true);
    },                                   ❺ Specifies redo action
    redo: function(){
        var action=null;
        if(this.curIdx < this.stack.length){
            action=this.stack[this.curIdx++];
        }else{
            action=this.getNewUndoAction();
        }
        action.canUndo=this.curIdx > 0;
        action.canRedo=this.curIdx < this.stack.length;
        this.undoHandler(action,false);
    },
    getNewUndoAction: function(theType,theValue){
        return {                         ❻ Creates undo action object
            type:   theType,
            value:  theValue,
            canUndo:false,
            canRedo:false,
            equals: function(action){
              return (
                JSON.stringify(this.type)  ==
                  JSON.stringify(action.type) &&
                JSON.stringify(this.value) ==
                  JSON.stringify(action.value)
              );
            }
        };
    },
};
```

The first thing that we do is initialize the stack ❶ by registering keyboard event handlers on the entire document. As with previous examples in this chapter, we need to bind the `keypress` event as well as the `keydown` event in order to stop event propagation in Mozilla. The main key handler method is `checkUndo()` ❷, which will be fired on all browsers, and captures Ctrl+Z (undo) and Ctrl+Y (redo) key bindings.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The init() method also requires a reference to a callback function, which we simply refer to initially. Once initialized, we can add items to the undo stack ❸. When doing this, we provide a default option to filter out duplicates. In many cases, if the new event matches the previous one, we won't want to add it to the list twice, although there are exceptions, as we'll see in section 8.3.3.

When an undo ❹ or a redo ❺ event occurs, we invoke this callback, passing in a small undo action object, which contains information on the state of the application and specifies whether the action can be undone or redone. Both undo() and redo() also perform a fair amount of bookkeeping to ensure that there is a previous or next action in the stack. If there's not, an empty object is returned, indicating that no action is possible.

The final thing to look at, then, is the structure of the action object itself ❻. The undo action encapsulates the data needed to perform an undo or a redo, as well as two Boolean values useful for determining whether another undo or redo action is available. It also provides a function for evaluating the equality of itself and another action object; add() makes use of this function to ensure that the same state isn't duplicated in two adjacent items in the stack.

Now that we have implemented the undo stack, let's create a sample undo handler to "undo-enable" the text input fields on our page. In the HTML for the page, we've defined two text inputs, with ids of foo and baz, respectively. Listing 8.8 shows the JavaScript required to hook these up to the undo stack.

Listing 8.8 Sample undo handler

```
window.onload = function(){
    addTextEventListener('foo');          ❶ Enables text inputs
    addTextEventListener('baz');
    undoStack.init(demoHandler);
}

function addTextEventListener(element){
    Event.observe(
        element,                  ❷ Captures keydown event
        'keydown',
        startTextUndo
    );
    Event.observe(
        element,                  ❸ Captures keyup event
        'keyup',
        endTextUndo
    );
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function startTextUndo(event){
    var target   =Event.element(event);
    var keyCode  =event.which || event.keyCode;
    if((keyCode != 16 && keyCode != 17 && keyCode != 18) &&
     !((keyCode ==  90) && event.ctrlKey) &&
     !((keyCode ==  89) && event.ctrlKey)){
        target.beforeUndoVal=target.value;      <-- 4 Updates previous value
    }
}

function endTextUndo(event){
    var target   =Event.element(event);
    var keyCode  =event.which || event.keyCode;

    if((keyCode != 16 && keyCode != 17 && keyCode != 18) &&
      !((keyCode ==  90) && event.ctrlKey) &&
      !((keyCode ==  89) && event.ctrlKey)){
        undoStack.add(                     <-- 5 Adds action to stack
            'TEXT',
            {   elementRef:target.id,
                prevValue :target.beforeUndoVal,
                newValue  :target.value
            }
        );
    }
}

function demoHandler(action,undo){
    if(action){
        if(action.type == 'TEXT'){
            var el=$(action.value.elementRef);
            el.value=(undo) ?              <-- 6 Updates input text
              action.value.prevValue :
              action.value.newValue;
        }

        $('undoButton').disabled=!action.canUndo;     7 Updates undo
        $('redoButton').disabled=!action.canRedo;       buttons
    }}
```

First, we enable both of our text inputs ❶ by binding event listener functions to them for both the `keydown` ❷ and `keyup` ❸ events. We need to capture both so that we can record the value before the keystroke on `keydown` ❹, and then add an undo action to the stack once the key is lifted again ❺. Note that we bypass certain keystrokes—Shift, Ctrl, and Alt—and the Ctrl+Z and Ctrl+Y keys, as we don't want to add these keypresses to the undo stack.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The undo action contains an identifier of the type of action—in our case we have used the label TEXT—and the value object, which must contain all the state needed to execute the action. In our case, we've recorded the current and previous contents of the element, as well as a reference to the element being updated. Our `undo()` handler method will be passed the undo action object. Once we've established that it is of the correct type, we can set the text input contents appropriately ❻, depending on whether it is an undo or a redo operation. We then enable or disable the on-screen undo and redo buttons appropriately ❼.

Discussion

This is only a generic implementation of an undo stack. Although it provides a reimplementation of the browser's undo functionality, no new features are added. For each custom undo operation, an action handler will need to be defined to handle the setting and unsetting of the data. We'll examine the implementation of such a custom operation in the next section.

Providing undo capabilities in your web application is a real win from a user experience standpoint. It helps to bridge the gap between regular nonbrowser applications and web applications. Many of the semantics that users are used to when dealing with native OS applications and thick clients can be reused and leveraged in your web application to reduce the user's learning curve. Granted, some web applications already have undo capabilities, but they are usually limited to a server-side implementation. Pushing this responsibility to the client makes web applications faster in their execution. This in turn makes the web application less frustrating for the user.

Previously we ran through some examples of maintaining history on both the client and the server. Naturally, this duality applies to undo functionality as well. In the next example, we'll take a look at how you can involve the server in the handling of the undo functionality.

8.3.3 Extending the undo stack for more complex actions

Now that we've seen a simple example of an undo stack, our engines are fully revved to handle a more complex example. Let's get to it, and at the same time involve the server, and Ajax, in our undo handling.

Problem

You need customized undo capability for actions that are more complex than simple text editing. Resetting the state of the application will require input from the server as well as the client.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

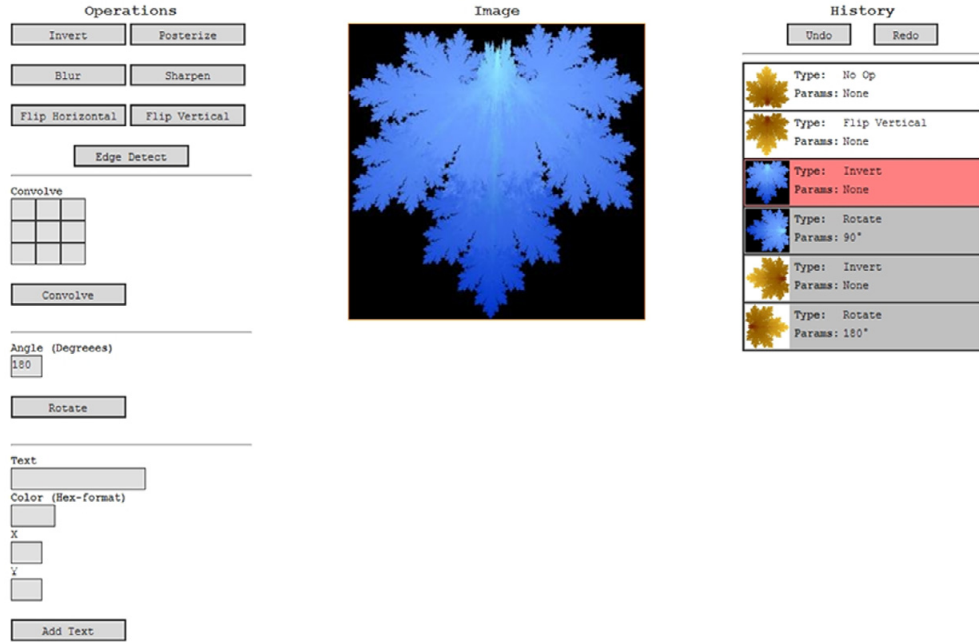

Figure 8.4 The completed image editor, combining a set of image-manipulation tools, a preview of the image, and a visual representation of the undo stack

Solution

This example will reuse our generic undo stack to provide undo capability for a simple image-manipulation application. The image operation actions will take place on the server, where a snapshot of the image's previous state will be stored, along with an operation reference and any other relevant operation data. The image editor user interface is shown in figure 8.4. The current state of the image appears in the middle, with a list of available operations on the left and a visual display of the undo stack on the right.

We first need to define the types of operations available in our image editor. These operations will be identified by a unique integer that will also be referenced by the server-side code. The operations available in the following example are as follows:

- `NO_OP` (no operation)—Used for the initial state
- `INVERT`—Takes the negative of the image's color values

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- POSTER (posterize)—Reduces the number of color levels in the image
- BLUR—Blurs the image using a 3x3 convolve matrix
- EDGE—Detects areas of the image with strong intensity contrasts
- SHARPEN—Reduces the "blurriness" of the image
- FLIP_H (flip horizontal)—Flips the image along the y-axis
- FLIP_V (flip vertical)—Flips the image along the x-axis
- ROTATE—Rotates the image (in degrees)
- STRING—Adds a text string to the image at the specified coordinate
- CONVOLVE—Allows the creation of a custom 3x3 convolve matrix

We'll need to write a lot of client-side code for this example, so let's break it down into stages. First, we define the image-manipulation operations as constants, as follows:

```
var NO_OP      = 0;
var INVERT_OP  = 1;
var POSTER_OP  = 2;
var BLUR_OP    = 3;
var EDGE_OP    = 4;
var SHARPEN_OP = 5;
var FLIP_H_OP  = 6;
var FLIP_V_OP  = 7;
var ROTATE_OP  = 8;
var STRING_OP  = 9;
var CONVOLVE_OP=10;
```

Next we initialize our generic undo stack object from the previous example. Remember that the stack took a callback function as an argument? We'll provide a callback here that will update the history palette on the right side of the screen and fetch the modified image from the server. Listing 8.9 shows how we've implemented this part of the program.

Listing 8.9 Setting up the complex undo stack

```
window.onload = initialize;

function initialize() {                          <-- 1 Creates initialize function
    undoStack.init(imageEditorHandler);          <-- 2 Initializes undo stack
    undoStack.doPrevActionCheck=false;           <-- 3 Ignores equality checks
    new Ajax.Request(                            <-- 4 Initializes server-side undo stack
      'ImageEditor?timestamp='+
        (new Date().getTime())+'&init=yes'
        +'&src='+encodeURI($('editingImage').src),
      { method: 'get',
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        onSuccess: addActionCallback
      }
    );}

function imageEditorHandler(action,undo) {        ❺ Specifies image-editor undo function
    if (action != null) {
        if (action.type != null) {
            var table = $('historyPalette');       ❻ Handles history palette state
            var selected=-1;

            for (var i = 0; i < table.rows.length; i++) {
                if(table.rows[i].selected) {
                    selected = i;
                    break;
                }
            }

            if (selected > -1) {
                if(undo){
                    table.rows[selected]
                      .className ='paletteDisabled';
                    table.rows[selected].selected  =false;
                    table.rows[selected].dimmed    =true;

                }else{
                    // set the next action
                    table.rows[selected]
                      .className ='paletteNormal';
                    table.rows[selected].selected  =false;
                    table.rows[selected].dimmed    =false;
                }

                var new_i=(undo) ? selected-1 : selected+1;
                table.rows[new_i].className='paletteSelected';
                table.rows[new_i].selected =true;
                table.rows[new_i].dimmed   =false;

                undoStack.seek(new_i);                 ❼ Activates arbitrary undo entry
                new Ajax.Request(
                  'ImageEditor?timestamp='+
                    (new Date().getTime())+'&seek='+(new_i),
                  { method: 'get',
                    onSuccess: seekCallback
                  }
                );
            }
        }

        $('undoButton').disabled=!action.canUndo;
        $('redoButton').disabled=!action.canRedo;
    }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function seekCallback(xhr){
    $('editingImage').src=xhr.responseText;
}
```

❽ **Defines Ajax callback**

The new `initialize()` function ❶ in listing 8.9 needs to do just a few things. We first need to initialize the client-side undo stack ❷ and give it a reference to the custom handler. Second, we need to tell the stack to ignore the equality check performed when adding a new undo action ❸. This will allow two image operations of the same type to be performed concurrently—for instance, blurring the image multiple times. Finally, we need to initialize the undo stack on the server side and set up the initial state of the application ❹.

The image-editor handler ❺ is slightly more complicated than in the previous example. When an actual action is performed, not only do we need to handle the undo/redo, but we also have to handle the state of the history palette ❻. Furthermore, with the introduction of the history palette, we need to be able to activate an arbitrary entry ❼ without first traversing the actions between the newly selected action and the previously selected one.

When we move to a new entry, we want to perform an Ajax request to the server to update the state of the server-side undo stack. We define a callback function for the Ajax request, which will simply update the main image source ❽. Although this is a simple callback, which we would normally write inline, we've defined it separately here so that we can reuse it elsewhere.

There's another predefined callback in this code as well, in the Ajax request that we make to initialize the stack. We'll introduce `addActionCallback()` shortly; it is somewhat more involved.

Let's move on, then, to look at the JavaScript behind the various actions that we have provided form controls for on the left. All of these actions follow a similar workflow, marshaling any necessary arguments, and making an Ajax.Request to the server, where the image manipulation will be performed. The function that creates this Ajax.Request is shown in listing 8.10.

Listing 8.10 `addAction()` method

```
function addAction(op){
    var action     =null;
    var paramString='';

    switch(op){
        case ROTATE_OP:{
            action={
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
                rotAngle: $F('rotAngle')
            };
            paramString='&params='+action.rotAngle;
            break;
        }
        case STRING_OP:{
            action={
                string: $F('string'),
                color:  $F('color'),
                locX:   $F('locX'),
                locY:   $F('locY')
            };
            paramString=
                '&params='+
                encodeURI(
                action.string.replace(/,/g,'%%')+','+
                action.color+','+
                action.locX+','+
                action.locY);
            break;
        }
        case CONVOLVE_OP:{
            action=[];
            for(var i=0;i<9;i++){
                action[i]=$F(
                    'c'+
                    Math.floor(i/3)+
                    '_'+
                    (i%3)
                );
            }
            paramString='&params='+encodeURI(action);
        }
        case INVERT_OP:
        case POSTER_OP:
        case BLUR_OP:
        case EDGE_OP:
        case SHARPEN_OP:
        case FLIP_H_OP:
        case FLIP_V_OP:
        case NO_OP:
        default:{break;}
    }

    undoStack.add(op,action);
    new Ajax.Request(
      'ImageEditor?timestamp='+
        (new Date().getTime())+'&action='+
        op+paramString,
      { method: 'get',
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            onSuccess: addActionCallback
          }
        );
    }
```

The first part of the function is a case statement that extracts any relevant arguments for a given operation into a variable called `paramString`. Note that several operations, such as invert and flip, require no arguments, whereas others, such as convolve and rotate, require several arguments. Having assembled the arguments, we update the client-side undo stack, and then make a call to the server, passing in the type of action along with any parameters. This will update the server-side undo stack.

Again, our Ajax request provides a reference to the `addActionCallback()` function. We've already seen this callback referenced in listing 8.9, when we initialized the stack. Let's take a look at what it does (listing 8.11).

Listing 8.11 `addActionCallback()` function

```
function addActionCallback(xhr){
    var table   =$('historyPalette');
    var content =xhr.responseText.split('<!-- BREAK -->');
    var found   =false;
    var toDelete=new Array();

    var editingImgSrc    =content[0];
    var newRowProperties=JSON.parse(content[1]);
    var eventCalls       =content[2];

    $('editingImage').src=editingImgSrc;

    for(var i=0;i<table.rows.length;i++){
        if(newRowProperties.id == table.rows[i].id){
            found=true;
        }

        if(found){
            toDelete[toDelete.length]=i;
        }else{
            table.rows[i].className='paletteNormal';
            table.rows[i].dimmed   =false;
            table.rows[i].selected =false;
        }
    }

    for(var i=toDelete.length-1;i>=0;i--){
        table.deleteRow(toDelete[i]);
    }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    var newRow     =table.insertRow(table.rows.length);
    var imageCell  =newRow.insertCell(0);
    var contentCell=newRow.insertCell(1);

    newRow.id       =newRowProperties.id;
    newRow.className='paletteSelected';
    newRow.selected =true;
    newRow.dimmed   =false;

    imageCell.className=newRowProperties.paletteImageClass;
    imageCell.innerHTML=newRowProperties.paletteImageHTML;

    contentCell.className=newRowProperties.paletteContentClass;
    contentCell.innerHTML=newRowProperties.paletteContentHTML;

    eval(eventCalls);
}
```

We've defined a custom format for the response returned by the server here, with a breaking delimiter splitting the response into three parts: the URL of the preview image, style properties defining the new entry in the history palette, and a set of JavaScript events to be called. We split the response body into these three sections, and then update the preview and the history palette accordingly.

We've now defined the full workings of the undo stack on the client and how it communicates with the stack on the server. The remaining JavaScript is simply concerned with adding some behavior to the history palette elements, as shown in listing 8.12.

Listing 8.12 Adding behavior to the history palette

```
function historyPaletteMouseOver(event) {            <-- 1 Highlights palette entry
    var tr = findTarget(Event.element(event));
    var id = tr.id;
    tr.className = 'paletteHighlight';
}

function historyPaletteMouseOut(event) {             <-- 2 Removes palette entry highlight
    var tr = findTarget(Event.element(event));
    var id = tr.id;
    if (tr.dimmed) {
        tr.className = 'paletteDisabled';
    } else if (tr.selected) {
        tr.className = 'paletteSelected';
    } else {
        tr.className = 'paletteNormal';
    }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function historyPaletteMouseDown(event) {      ❸ Selects palette entry
    var tr = findTarget(Event.element(event));
    var id = tr.id;
    tr.className = 'paletteSelected';
}
                                              ❹ Disables later entries
function historyPaletteMouseClick(event) {
    var table  = $('historyPalette');
    var tr     =findTarget(Event.element(event));
    var id     =tr.id;
    var disable=false;
    for(var i = 0; i < table.rows.length; i++) {
        var className = 'paletteNormal';
        table.rows[i].dimmed   = false;
        table.rows[i].selected = false;
        if (disable) {
            table.rows[i].dimmed = true;
            className = 'paletteDisabled';
        } else if (tr.id == table.rows[i].id) {
            table.rows[i].selected = true;
            className = 'paletteSelected';
            disable = true;
        }

        table.rows[i].className = className;
    }

    var idx = parseInt(id.replace(/action_/,''));

    undoStack.seek(idx);
    seekRequest=sendGETRequest(
        '/servlet/ImageEditor?timestamp='+
        (new Date().getTime())+'&seek='+
        idx,seekCallback
    );
}
                                    ❺ Activates specific
function findTarget(element) {         undo action
    var parent = null;

    while (parent == null) {
        if (element.id.indexOf('action_') == 0) {
            parent = element;
        } else {
            element = element.parentNode;
        }
    }

    return parent;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The history palette needs several events to handle its look and feel as well activate a specified undo action ❺. We will need event handlers for

- `onMouseOver` to highlight the palette entry ❶
- `onMouseOut` to un-highlight the palette entry ❷
- `onMouseDown` to select the palette entry ❸
- `onClick` to select the palette entry, activate the undo stack, and disable all later entries ❹

Discussion

This example has provided a much richer undo stack. You can use this example to create your own action handlers that provide undo capability for just about anything that a user does. You now have an undo/redo framework with a history palette to build on.

A new problem you run into when maintaining undo data on the server is one of garbage collection. When is it safe to delete the undo information for a user? This is a question you must decide based on the merits of your application and the functionality you wish to present to users. One could simply expire the undo information when a user's session times out. That is not difficult to implement, as many web application frameworks provide callbacks that are invoked on session timeout. Of course, in this case your users will not be able to stop working in one browser and resume working in another. If you wish to provide users with that type of functionality, then some concept of a workflow must be created on the server and tied to a user. Once that is in place, users can resume their workflow at the point where they left off. If you do decide to move to a workflow-oriented undo stack, then you should think hard about when it is safe to expire a user's undo stack and inform users of the undo stack behavior.

Related to this, you also need to decide how much of a history you're willing to maintain. In the case of our image-editing program, users could abuse our server and use up all of our storage space if we failed to limit their undo stacks. Simply by applying a lot of image transforms (perhaps from an automated script), they could exhaust server resources by creating huge undo stacks.

8.4 Summary

In this chapter we've seen how it is possible to deprive the end user from navigating history or refreshing a page. We've also seen, through the use of hashes and the Really Simple History framework, how to work with the history and

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

refresh features of the browser. It is much easier to just deny this functionality to the end user, but such an approach would ultimately drive some people away from your application. The smart way to address the issue is to work with these features and reimplement client history to provide a rich and usable experience for the end user.

We have also implemented a simple undo stack to allow you, the developer, to create undoable actions for just about anything the user can do, not just for filling out forms. This increases the flexibility of your application, and gives some control back to users when they make mistakes or simply change their minds about an action.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

9 Drag and drop

This chapter covers

- Drag and drop basics
- Drag and drop lists
- ICEfaces drag and drop

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We're all familiar with "drag and drop" in desktop applications; in fact, drag and drop is a key part of what gives an air of reality to our interactions with computers. ("My data isn't just a bunch of zeros and ones—I have files and folders that move when I touch them; they're real.") The most familiar application is probably in the file browser. Users are presented with icons for files and folders; users can open folders to display their contents in a window and—most interesting to us as developers—they can drag icons (by pressing the mouse button and moving the mouse) from one location and drop them (by releasing the mouse button) in another. It's an important way of saying, "Do something with this to that." Other user interface techniques (such as clicking a button or selecting a menu) typically just let you say, "Do something to this." Look at figure 9.1.

We've just explained drag and drop in terms of its primitive mouse events, but this is not necessarily how it appears to the desktop application developer. Instead, it typically appears as a data transfer operation, with abstract drag-and-drop events, and the concept of data being moved from one object to another. In other words, drag and drop is often tightly integrated with copy and paste. We won't be taking drag and drop quite this far in our treatment in this chapter, but we'll find drag-and-drop events that are sufficiently abstract to comfortably build some very interesting applications.

What's exciting about drag and drop is the fact that even though it's been around since the beginning of desktop graphical interfaces, a lot of applications still don't make good use of it. In other words, the techniques that we'll learn in this chapter present an opportunity to make Ajax applications that are actually

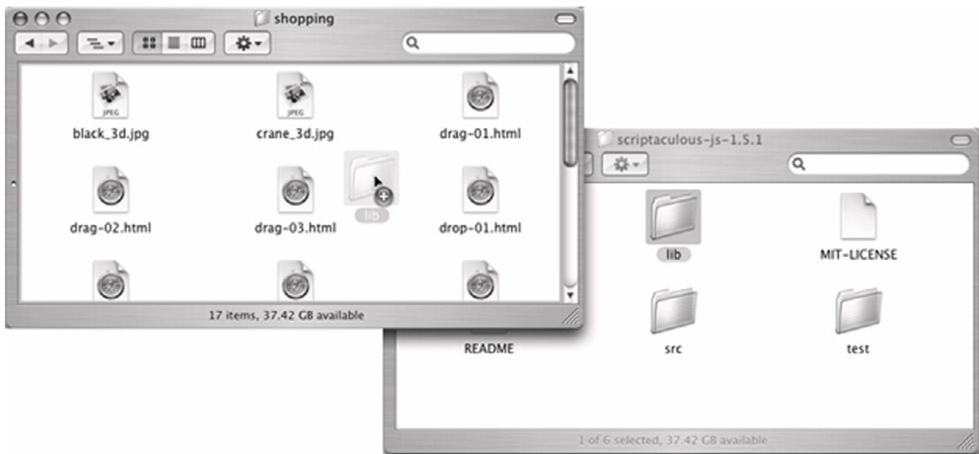

Figure 9.1 Desktop drag and drop

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

superior to their desktop counterparts. As you might expect, implementing drag and drop in a browser takes a bit of JavaScript magic. Fortunately, it's JavaScript that various nice people have already implemented. Let's move on and see what JavaScript drag-and-drop frameworks are available to us.

9.1 JavaScript drag-and-drop frameworks

There are a number of open source JavaScript implementations of drag and drop, but it's important to choose one that is being actively maintained (to be assured that it will run reliably on all modern web browsers) and that has an API that is easily usable with Ajax. Two of the more popular frameworks are Rico (http://openrico.org) and Script.aculo.us (http://script.aculo.us/).

Rico can be used to create rich Internet applications and provides full Ajax support, drag-and-drop management, and a cinematic effects library. Script .aculo.us also provides cinematic visual effects and an API for drag and drop. One thing that we won't be doing is dragging and dropping between other applications and the browser. As powerful as these libraries are, anything that we make draggable is strictly confined to the browser window.

Using the two APIs is really quite similar. For either API, the objects that can be dragged and the places they can be dropped are simply HTML `<div>`s. In HTML we just need something like the following:

```
<div id="dragster">
  Drag This
</div>

<div id="dropster">
  Drop Something Here
</div>
```

But nothing's draggable yet. We need to register the `<div>`s with the framework of our choice by passing in their `id`s. Using Rico, this would be

```
<script type="text/javascript">
    dndMgr.registerDraggable(
        new Rico.Draggable(
          "test-draggable", "dragster"));
    dndMgr.registerDropZone(new Rico.Dropzone("dropster"));
</script>
```

And using Script.aculo.us,

```
<script type="text/javascript">
    new Draggable("dragster")
    Droppables.add("dropster");
</script>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

So, either framework is very easy to use. Since the techniques for one can readily be adapted for the other and Script.aculo.us shows slightly better browser compatibility, the examples in this chapter will feature Script.aculo.us. Next, let's see how to mix Ajax into the draggable (things that can be dragged) and droppable (where draggables can be dropped) objects we've created.

9.2 Drag and drop for Ajax

Clearly, we are not going to implement drag-and-drop functionality from scratch in JavaScript. By making use of a library, such as Script.aculo.us, we can ensure that the drag-and-drop features in our application not only work correctly but also are portable across a variety of browsers. For us, the challenge simply becomes how to tie drag-and-drop features in with Ajax. Without Ajax, all we can do is provide a little toy for users: they can drag some items around in their browser and be amused with the novelty of this activity, but there's no way for them to share what they've done with other users or affect the real data that lies on the server. To make the application real, we need to hook the drag-and-drop events into Ajax calls to the server. Let's see how to do this with a shopping cart example that makes use of Script.aculo.us.

9.2.1 Drag-and-drop Ajax shopping cart

By applying drag and drop, we can provide a shopping cart that is intuitive for users, allowing them to drag items that they see on the screen into a shopping cart for purchase. Three items will be available for purchase, as shown in figure 9.2: two books and a rock. If the user shouldn't purchase a particular item (perhaps because they've run out of money, or, in our particular case, because the item is a rock and not a book), the shopping cart can reject the item, causing it to revert to its original position.

The concept is simple, but there are a number of user interface events that need to be brought to life with Ajax: `onHover`, `onDrop`, and `revert`. Since our application has only one type of object that can be dragged (items for purchase) and only one place for the objects to be dropped, the events have the following meanings:

- `onHover`—The user may be considering the item; we might as well encourage them to buy it. It can also indicate an item in the process of being removed from the shopping cart.
- `onDrop`—The user has selected an item for purchase.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- `revert`—The `revert` callback is our application's chance to say whether the object should drop in place or snap back to where it came from. We return `true` for `revert` when we reject an item from the shopping cart.

We now have defined our draggable regions (the books and the rock), our droppable regions (just the shopping cart), and our events (`onHover`, `onDrop`, and `revert`). Let's take a look at the implementation.

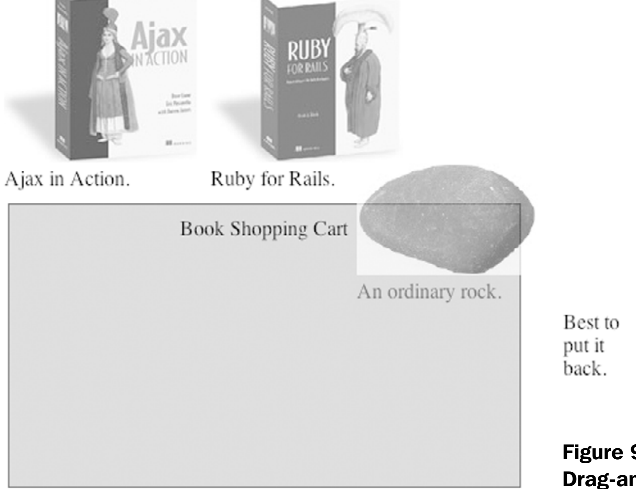

Figure 9.2
Drag-and-drop shopping cart

Problem

You need browser drag-and-drop events to be communicated to the server.

Solution

To understand this example, we'll begin by looking at the HTML; once we understand that, the role of the JavaScript will be clear. In the HTML that follows, we'll find a series of major pieces:

- JavaScript library loading
- Style definitions
- `<div>` elements for our draggable and droppable regions
- A `<span>` for our status message
- Registration of the droppable region with Script.aculo.us

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

There's only one more detail to keep in mind: each draggable region will be registered with Script.aculo.us when it's defined. Depending on how you like to organize your code, you can take this approach, or register all the draggable regions together in one block. Now, if we didn't need to do any Ajax, and if we were content to just let the user drag objects around in the browser and not send any information back to the server, this would be all we need. As you may suspect, we'll apply the Ajax when we get to the JavaScript for this example. Let's get started by looking at the HTML in listing 9.1.

Listing 9.1 Shopping cart HTML

```
<html>
<head>

<script type="text/javascript" src="lib/prototype.js"></script>
<script type="text/javascript" src="lib/scriptaculous.js"></script>

</head>

<style type="text/css">
  div.carthoverclass {
    border:1px solid blue;
  }
  div.cart  {
    z-index:100;
    text-align:center;
    height:200px;
    padding:10px;
    background-color:#abf;
  }
</style>

<body>
  <h3>Ajax Shopping Cart</h3>

  <table><tr>

    <td>
      <div alt="Product1" id="product_1"
         itemid="01" style="z-index:500" />
        <img alt="Product1" src="crane_3d.jpg" />
        <br />
        Ajax in Action.
      </div>
      <script type="text/javascript">
        new Draggable('product_1',
            {revert:handleRevert});
      </script>
    </td>
```

Annotations: Loads JavaScript libraries (script tags); Specifies cart border highlight style (div.carthoverclass); Specifies shopping cart style (div.cart); Defines first draggable item (div product_1); (1) Registers first draggable item (new Draggable).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      <td>
        <div alt="Product2" id="product_2"
             itemid="02" style="z-index:500" />
          <img alt="Product2" src="black_3d.jpg" />
          <br />
          Ruby for Rails.
        </div>
        <script type="text/javascript">
        new Draggable('product_2', {revert:handleRevert});
        </script>
      </td>

      <td>
        <div alt="Product3" id="product_3"
             itemid="03" style="z-index:500" />
          <img alt="Product3" src="rock.jpg" />
          <br />
          An ordinary rock.
        </div>
        <script type="text/javascript">
          new Draggable('product_3', {revert:handleRevert});
        </script>
      </td>

    </tr></table>

    <table><tr>

      <td width="400px">
        <div id="cart" class="cart" >
          Shopping Cart
        </div>

      </td>
      <td width="25">
      </td>

      <td width="50">
        <span id="cartinfo"></span>
      </td>

      </tr></table>

  <script type="text/javascript">
    cartinfoDiv = $("cartinfo");
    Droppables.add('cart',
          { hoverclass:'carthoverclass',
            onHover:
            function(dragged, dropon, event) {
                   handleHover(dragged, dropon,
                           event, cartinfoDiv);
            },
```

Annotations: Defines second draggable item; Defines third draggable item; Defines droppable; Defines status message region; (2) Registers shopping cart droppable

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          onDrop: :
          function(dragged, dropon, event) {
                 handleDrop(dragged, dropon,
                         event, cartinfoDiv);
    }
        }
   )
</script>

</body>
</html>
```

❷ Registers shopping cart droppable

We pass the id of our draggable item <div> to Script.aculo.us ❶ and register a callback for revert so we can control whether an item stays in the shopping cart or snaps back. We also pass the id of our droppable shopping cart <div> ❷ to Script.aculo.us and register onDrop and onHover anonymous callbacks so that we can pass the id of our status message region.

As you see, draggable and droppable <div>s can contain anything—text, images, and so on—so it's just a matter of creating the appropriate HTML to make the objects look the way you want. Once you have the <div>s, they just need to be registered with Script.aculo.us as draggable or droppable, and this will cause Script.aculo.us to animate the items appropriately when the user drags them with their mouse. Of course, we also registered a number of callbacks as well; the callbacks are the event handlers for when a draggable is moved over a droppable (onHover) or when it's dropped on a droppable (onDrop). We also registered a revert callback that lets us return true or false to indicate whether the item should stay in the shopping cart or should snap back to its original location. What will we do in these callbacks? Ajax. Two main Ajax functions are performed by the scripting: fetching new status messages depending on how the user has moved the draggable item (such as thanking them for the purchase when they drop the item into the shopping cart), and fetching the revert status of the draggable item they're using (the rock can't be purchased, so it must snap back to its starting location). The rest of the script just makes sure that our user interface events get delivered correctly—which turns out to be a bit tricky for the hover events. Figure 9.2 shows what the HTML produces, and listing 9.2 contains the HTML itself.

Listing 9.2 Shopping cart JavaScript

```
<script type="text/javascript">
function AjaxHTML(url, target)  {
    AjaxOperation(url, function(req) {replaceHTML(req, target);});
}
```

Updates HTML via Ajax

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function AjaxOperation(url, func, asynch) {
   var req;
   if (window.XMLHttpRequest) {
       req = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
       req = new ActiveXObject("Microsoft.XMLHTTP");
   }
   if (req) {
       req.onreadystatechange =
           function (){onReady(req, func);};
       req.open("GET", url, true);
       req.send("");
   }
}

function onReady(req, func) {
   if (req.readyState == 4) {
       if (req.status == 200) {
           func(req);
       }
   }
}

function replaceHTML(req, target)  {
    var content = req.responseText;
    target.innerHTML = content;
}

var revertFlag = true;
function checkRevert(req)  {
    var content = req.responseText;
    revertFlag = (1 == (content - 0));
}

var lastHover;
function handleHover(dragged, dropon, event, target)  {
    itemid = dragged.getAttribute("itemid");
    if (lastHover == itemid)  {
        return;
    }
    lastHover = itemid;
    AjaxOperation("revert-" + itemid + ".txt",
            function(req) {checkRevert(req)});
    AjaxHTML("hover-" + itemid + ".html", target);
}

function handleDrop(dragged, dropon, event, target)  {
    lastHover = null;
    itemid = dragged.getAttribute("itemid");
    AjaxHTML("drop-" + itemid + ".html", target);
}
```

Annotations:
- ❶ (the XMLHttpRequest / ActiveXObject creation block)
- ❷ (the onreadystatechange assignment)
- Waits for complete asynchronous response (onReady)
- Updates HTML with Ajax response (replaceHTML)
- ❸ (checkRevert body)
- ❹ (the lastHover check)
- ❺ (the AjaxOperation revert call)
- Asynchronously shows hover status via Ajax
- ❻ (lastHover = null;)
- Asynchronously shows drop status via Ajax

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
function handleRevert(dragged)  {
    return revertFlag;
}
</script>
```

Returns revert flag for Script.aculo.us

Discussion

Now that we've seen the code, let's discuss some of the more interesting bits a little further. Because we need to perform multiple Ajax requests in rapid succession, we maintain distinct request objects ❶ when we create our callback ❷. This keeps them from overwriting each other. It's also important to note that the `revert` flag is encoded as 0 for `false` and 1 for `true`. Here ❸ we extract it from the Ajax response and convert it into a JavaScript Boolean value. Also keep in mind that every mouse movement generates a hover event, so we filter out ❹ all but the first one. After we perform our filtering, we fetch the `revert` flag synchronously ❺ before the hover message is fetched. We do this so that the two requests do not interfere with each other. Finally we reset `lastHover` ❻. This ensures that the `handleHover()` function will be fully executed when it is called next. We do this because, once we drop an item, we might drag it again.

For the most part, the JavaScript in listing 9.2 simply responds to events and fetches page updates from the server via Ajax. It's only in the `handleHover()` where we see a bit of complication. The first problem is that `handleHover()` isn't just called when the hover starts; it's called for every movement of the mouse while the draggable is hovering over the droppable. There's a slim possibility that you want to send every mouse movement over the network, but it's not likely, so we've added logic to process only the first hover event. This works by keeping track of the draggable currently hovering; if it's the same draggable as last time, just return from `handleHover()` without further processing.

The second problem is that we need to perform two operations in `handleHover()`: update the display with a relevant message encouraging the user to purchase, and check whether or not the item should be reverted from the shopping cart. Often, simple Ajax applications can get away with a single, global, request object, but in this case we need to guarantee that we process both the `revertFlag` and the hover message correctly. The solution is to pass the XMLHttpRequest object as a parameter so that each different request can be acted on individually.

A real shopping application would likely have more than three items to choose from, and the items would probably be dynamically generated. Fortunately, it's just a matter of including the registration code following each item:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
<script type="text/javascript">
  new Draggable('product_1', {revert:handleRevert});
</script>
```

9.2.2 Manipulating data in lists

The basic drag-and-drop primitives of draggables and droppables are very powerful, so it's easy to imagine how a variety of applications could be built up. For instance, what if you have some items, say, fruits and vegetables, and you want to organize them into two lists: by type (fruit or vegetable) and by preference (by their order in the lists). A drag-and-drop interface makes sense; all the items should be draggable (so you can move them around), and both of the lists should be droppable. When you drag an item from one list to the other, it is removed from the first list and added to the second list. When you drag an item within a list, the other items should move out of the way so that the item can be dropped in a particular place. This would indeed require a complex chunk of JavaScript in the `onHover` event handler. Fortunately, Script.aculo.us provides built-in primitives for manipulating lists with drag and drop. All that we need to do is to create the lists in HTML using the HTML list tags, and then indicate to Script.aculo.us which items can be used in which lists and what to do when the lists have been changed. When the lists change, it's our chance to apply Ajax and invoke our application on the server, as shown in figure 9.3.

For this problem, we'll just inform the user of the first fruit or vegetable that they've been misplaced. Since we need to pass a relatively complex structure to

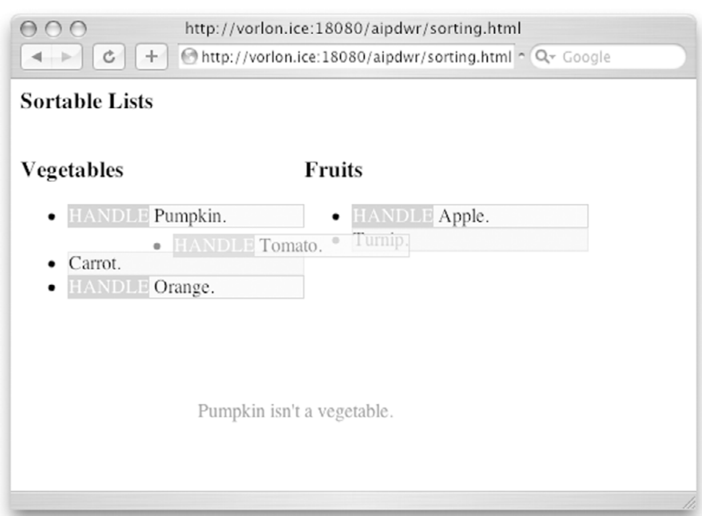

Figure 9.3
Verifying vegetables

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the server (two list orderings), we'll be using DWR to handle the data marshaling. Between Script.aculo.us and DWR, we are able to make use of these powerful frameworks to concentrate our effort on the server-side Java code. As you'll see, the trickiest part is determining what is a fruit and what is a vegetable.

Problem

You want to provide lists of data that can be modified with drag and drop.

Solution

Like most Ajax applications, our code will be divided into three main pieces: HTML, JavaScript, and application code. Since we're applying DWR and Script .aculo.us, our HTML and JavaScript will be fairly simple—essentially registration code and event handlers, and our application code will be in Java. Let's start with the HTML (listing 9.3) and build the two lists.

Listing 9.3 List HTML

```
<html>
<head>
<style type="text/css">
li.green {
  background-color: #ECF3E1;
  border:1px solid #C5DEA1;         Specifies vegetable style
  cursor: move;
}

li.orange {
  border:1px solid #E8A400;         Specifies fruit style
  background-color: #FFF4D8;
}

span.handle {
  background-color: #E8A400;        Specifies fruit
  color:white;                      handle style
  cursor: move;
}
</style>
</head>
<body>

<h3>Sortable Lists</h3>

<div style="height:200px;" >

<div style="float:left;" >
<h3>Vegetables</h3>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  <ul id="vegetables"
      style="height:150px;width:200px;">
    <li class="orange" id="left_Pumpkin">
      <span class="handle">HANDLE</span> Pumpkin.
    </li>
    <li class="green" id="left_Carrot">Carrot.</li>
    <li class="orange" id="left_Orange">
      <span class="handle">HANDLE</span> Orange.
    </li>
  </ul>
</div>

<div style="float:left;" >
  <h3>Fruits</h3>
  <ul id="fruits" style="height:150px;width:200px;" >
    <li class="orange" id="right_Apple">
      <span class="handle">HANDLE</span> Apple.
    </li>
    <li class="orange" id="right_Tomato">
      <span class="handle">HANDLE</span> Tomato.</li>
    <li class="green" id="right_Turnip">Turnip.</li>
  </ul>
</div>

<br>

<div style="clear: both; margin-left: 150px;" >
  <span id="plant-error"
        style="color: red;" ></span>
</div>

</div>

<script type="text/javascript">
  Sortable.create("vegetables",
     {dropOnEmpty:true,
     containment:["vegetables","fruits"],
     constraint:false,
     onUpdate:handleUpdate});
   Sortable.create("fruits",
     {dropOnEmpty:true,handle:'handle',
     containment:["vegetables","fruits"],
     constraint:false,
     onUpdate:handleUpdate});
</script>

</body>
```

Contains left list of assorted items

Contains right list of assorted items

Specifies application message region

Registers left list

Registers right list

This gives us our lists of items, a place to show application messages, and Script.aculo.us registration calls so that the items can be dragged and dropped.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Next, we need to apply a layer of DWR to glue all of this to our application on the server. We simply need to include the required DWR libraries and implement the Script.aculo.us callbacks with DWR functions (listing 9.4).

Listing 9.4 List JavaScript

```
<head>
<script type="text/javascript" src="lib/prototype.js">
  </script>
<script type="text/javascript" src="lib/scriptaculous.js">
  </script>
<script type="text/javascript" src="dwr/engine.js"></script>
<script type="text/javascript" src="dwr/util.js"></script>
<script type="text/javascript" src="dwr/interface/Demo.js"></script>

<script type="text/javascript">
function handleUpdate(list)  {
    Demo.checkPlant(
            Sortable.serialize("vegetables")
            + "&" + Sortable.serialize("fruits"),
             showPlantMessage);
}

function showPlantMessage(message)  {
    DWRUtil.setValue("plant-error", message);
}
</script>
</head>
```

Loads Script.aculo.us libraries
Loads DWR base libraries
Loads DWR application library
❶ Encodes changed lists for application
❷ Displays application message

There are two lists, and either one can change, so ❶ we send the current states of both lists to the server upon any change. The server will respond to our list update ❷ with a message that we insert into the page with `DWRUtil.setValue()`.

As can be seen in the DWR script functions, we only have one server-side method to implement: based on the content of the two lists, we'll return a String that indicates if there is an error (listing 9.5).

Listing 9.5 List Java

```
package aip;

import java.util.HashMap;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
public class Demo {

    Map plants;

    public Demo() {
        initPlantData();
    }

    private void initPlantData() {
        plants = new HashMap();
        List fruits = new ArrayList();
        List vegetables = new ArrayList();
        fruits.add("Apple");
        fruits.add("Pumpkin");
        fruits.add("Orange");
        fruits.add("Tomato");

        vegetables.add("Carrot");
        vegetables.add("Turnip");

        plants.put("vegetables", vegetables);
        plants.put("fruits", fruits);
    }

    public String checkPlant(String listUpdate) {
        String[] items = listUpdate.split("&");
        for (int i = 0; i < items.length; i++) {
            if ("".equals(items[i])) {
                continue;
            }
            String[] pair =
                items[i].split("\\[\\]=");
            List plant = (ArrayList) plants.get(pair[0]);
            if (!plant.contains(pair[1])) {
                return pair[1] + " isn't a " +
                pair[0].substring(
                    0, pair[0].length() - 1) +
                ".";
            }
        }
        return "";
    }
}
```

Initializes fruit/ veg expert system

Splits encoded list into items

Splits items into list/entry

Renders verdict on fruit/veg

Let's look at the `checkPlant()` method in more detail because it has very little to do with plants but has a lot to do with decoding Script.aculo.us serialized lists. In our Script.aculo.us callback `handleUpdate()`, we append the serialization of the second list to the first and send it to the server via DWR. The `listUpdate` parameter might look something like this:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
vegetables[]=Pumpkin&vegetables[]=Carrot&vegetables[]=Orange
        &fruits[]=Apple&fruits[]=Tomato&fruits[]=Turnip
```

Individual items are separated by `&` (just like HTML form values) and the individual items are encoded as follows:

```
listname[]=id
```

So, to decode them, we split the entire string at each `&` into items, and then split each item into its list name and its `id` at each `[]=`. The rest of the `checkPlant()` method just looks up the items in the database (not a very large database, in this case) to find the first error. When it finds an error, the method generates an informative message based on the data.

Discussion

Script.aculo.us makes it easy to develop an application that lets the user work with lists. Unlike with a desktop API, though, the changes to the lists are not passed to us as objects. Instead, we receive the current state of the list as a sequence of `id`s. For this reason, it's important to assign meaningful `id`s to the list elements, and to not merely assign them `id`s based on their position in the list; more specifically, the `id` we use should uniquely identify the list item in our application. There are a variety of possibilities, but you may wish to use a database key or an object hash code; it can be anything that can be uniquely resolved once it reaches the server (but it should be reasonably short, because the entire list of `id`s will be sent).

Though Script.aculo.us is an amazing tool, it's not the only one in our toolbox. Let's next take a look at another popular Ajax framework, ICEfaces, which you can use to implement drag-and-drop capabilities in your Ajax applications.

9.2.3 *The Ajax shopping cart using ICEfaces*

ICEfaces is an open source toolkit for developing Ajax applications as standard JavaServer Faces (JSF) applications (http://www.icefaces.org). The distinguishing characteristics of ICEfaces are the development methodology and the natural Ajax Push application-initiated update capability. (Ajax Push is closely related to "Comet" or "Reverse Ajax" and can be used to implement notifications or multiuser collaboration features within web applications.) Developing an ICEfaces application is just a matter of developing a standard JSF application. The application can become an Ajax application with no code changes, thereby preserving the strong Model-View-Controller separation emphasized by JSF. To benefit from application-initiated updates, a single ICEfaces API method is all that is required;

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the application simply calls `render()` on the server when a page should be updated and the ICEfaces framework determines and pushes to the browser the minimal page updates required.

To get an idea of how to develop an application in ICEfaces, let's use it to implement the drag-and-drop shopping cart example that we previously put together with JavaScript and Script.aculo.us. It's a simple application: the user can drag any of the three items and the application will update the message beside the shopping cart depending on which item they drag and whether they drop it in the cart.

Problem

You want to implement a drag-and-drop shopping cart using ICEfaces.

Solution

Take a look at the Ajax shopping cart shown in figure 9.4. If you think it looks and feels familiar, you're right; ICEfaces makes use of the Script.aculo.us libraries to provide drag and drop—the JavaScript interface to Script.aculo.us is just abstracted for the Java developer behind a JSF component model provided by ICEfaces.

Our implementation will be in three main sections: a user-interface declaration (which is expressed in listing 9.6 as a JSP document but strongly resembles HTML markup), two straightforward JavaBeans, and an XML configuration file

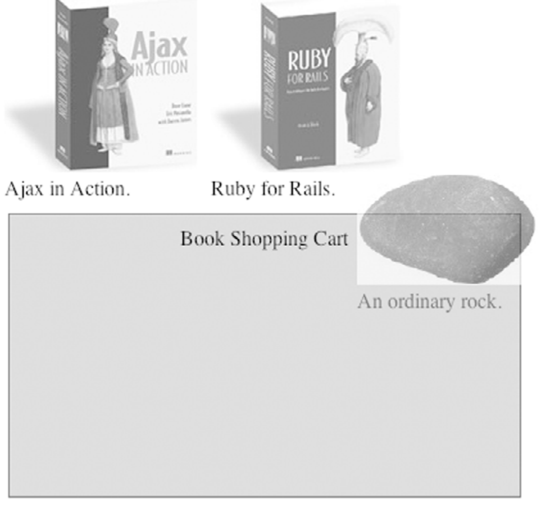

Figure 9.4
The drag-and-drop shopping cart, now with ICEfaces

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

containing injected application data. Let's dive into the markup, where we're going to see a small amount of JSF configuration and a CSS style, but mostly we'll devote our attention to configuration of the draggable components. The other part to watch for is how the shopping cart message is displayed; it's so simple that it's easy to miss, as it's nothing more than a text output component bound to a JavaBean. ICEfaces takes care of all the Ajax for us; when an item is dropped and the message changes, the minimal page update is determined and automatically applied. We just need to specify how the parts of our page are bound to the data in our model.

Listing 9.6 ICEfaces JSF

```
<f:view xmlns:f="http://java.sun.com/jsf/core"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:ice="http://www.icesoft.com/icefaces/component">   Declares JSF tag namespaces

 <html>
 <head>
    <style type="text/css">
      div.cart  {
        z-index:100;
        text-align:center;
        height:200px;                                Contains HTML
        padding:10px;                                markup and
        background-color:#abf;                       CSS style
      }
    </style>
 </head>
 <body>

  <h3>Ajax Shopping Cart</h3>

  <ice:form>          <-- Specifies form containing
                          input components
   <table><tr>

    <td>
      <ice:panelGroup                                     Defines container for
          style="z-index:500;cursor:move;"                draggable book
          draggable="true"    <-- Indicates draggable
          dragListener=                                   (1)
            "#{dndBean.dragListener}"
          dragMask=                                       (2)
            "dragging,drag_cancel,hover_end"
          dragOptions=                                    (3)
            "#{craneItem.dragOptions}"
          dragValue="#{craneItem}" >   <-- (4)
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <img alt="Product1" src="crane_3d.jpg" />       Specifies HTML for
        <br/>                                           draggable book
          Ajax in Action.
      </ice:panelGroup>          <-- Closes draggable
                                     container
    </td>

    <td>
      <ice:panelGroup
          style="z-index:500; cursor:move;"
          draggable="true"
          dragListener=
            "#{dndBean.dragListener}"
          dragMask=
            "dragging,drag_cancel,hover_end"          Specifies second
          dragOptions=                                draggable book
            "#{blackItem.dragOptions}"
          dragValue="#{blackItem}">
        <img alt="Product2" src="black_3d.jpg" />
        <br/>
          Ruby for Rails.
      </ice:panelGroup>
    </td>

    <td>
      <ice:panelGroup
          style="z-index:500; cursor:move;"
          draggable="true"
          dragListener=
            "#{dndBean.dragListener}"
          dragMask=
            "dragging,drag_cancel,hover_end"      Specifies
          dragOptions=                            draggable rock
            "#{rockItem.dragOptions}"
          dragValue="#{rockItem}">
        <img alt="Product3" src="rock.jpg" />
        <br/>
          An ordinary rock.
      </ice:panelGroup>
    </td>

  </tr></table>

  <table><tr>

    <td width="400px">
      <ice:panelGroup style="z-index:0;"      Specifies HTML for
          dropTarget="true">                  cart drop region
        <div id="cart" class="cart" >
            Book Shopping Cart              Defines drop target
        </div>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      </ice:panelGroup>
    </td>

    <td width="25">
    </td>

    <td width="50">
      <ice:outputText
          value="#{dndBean.dragMessage}"/>      Contains message text
    </td>                                       from JavaBean

   </tr></table>

  </ice:form>
 </body>
 </html>
</f:view>
```

dndBean ❶ is the name of the JavaBean modeling our shopping cart application. It has a listener method called dragListener() that this binding expression tells JSF to call when something interesting happens with this draggable item.

We're not interested in every event relating to drag and drop. ICEfaces handles the events on the server ❷, and we don't want to send each mouse movement from the dragging action to the server, so we use this mask to block the events that are not of interest.

We want the different items to have different dragging behavior (such as whether the item should snap back when dropped). By binding this behavior to the item bean with an expression ❸, we can configure it individually for each item.

The dragItem is an actual JavaBean object associated with the draggable item that our dragListener() method can operate on when drag and drop occurs. This lets our Java application work with objects that are meaningful at the application level ❹.

Were you able to spot the Ajax in the declaration of the page? It's intentionally invisible—one of the goals of ICEfaces is that Ajax be transparently provided. The developer focuses on the application: what components go on the page and how those components are bound to the dynamic data model. The low-level aspects of how those components are updated over the network using XHR are all abstracted and handled by the framework.

We've created a page that declares what components are displayed to the user; now we need the executable part of our application. This will be implemented

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

entirely on the server in JavaBeans. Let's start with the simplest part of our model: the shopping items. ShoppingItem (listing 9.7) is purely data-oriented; it's just a JavaBean capable of containing information about the item. In a more complete application, we would also include such things as the price, weight, and availability, but for this simple example our item will just contain the messages to show when the user drags it and drops it, and the options to use for dragging behavior (so just the rock can snap back rather than dropping into the shopping cart).

Listing 9.7 ShoppingItem

```
package aip;

import java.io.Serializable;

public class ShoppingItem implements Serializable  {

    String hoverMessage = "";

    public void setHoverMessage(String message) {
        this.hoverMessage = message;
    }

    public String getHoverMessage() {
        return this.hoverMessage;
    }

    String dropMessage = "";

    public void setDropMessage(String message) {
        this.dropMessage = message;
    }

    public String getDropMessage(){
        return this.dropMessage;
    }

    String dragOptions = "";

    public void setDragOptions(String options) {
        this.dragOptions = options;
    }

    public String getDragOptions() {
        return this.dragOptions;
    }

}
```

Sets/gets hover message

Sets/gets drop message

Sets/gets drag options

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

It's easy to see that ShoppingItem can represent the properties of a shopping item in our simple application, but how do the individual items get created and how do they get configured? An interesting technique is to use the dependency injection or inversion of control capability with managed beans. In this example (listing 9.8), we'll use managed beans merely to instantiate and specify the items in our application, injecting the beans into the user's session from the outside (this simple application has no meaningful dependencies).

Listing 9.8 ICEfaces JSF configuration

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

    <managed-bean>
      <managed-bean-name>dndBean</managed-bean-name>
      <managed-bean-class>aip.DragDropBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>craneItem</managed-bean-name>
      <managed-bean-class>aip.ShoppingItem</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>hoverMessage</property-name>
            <value>Good choice.</value>
        </managed-property>
        <managed-property>
            <property-name>dropMessage</property-name>
            <value> Thank you for your purchase.</value>
        </managed-property>
        <managed-property>
            <property-name>dragOptions</property-name>
            <value></value>
        </managed-property>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>blackItem</managed-bean-name>
      <managed-bean-class>aip.ShoppingItem</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>hoverMessage</property-name>
            <value>Excellent selection.</value>
        </managed-property>
```

Annotations:
- Declares JSF config file
- Declares drag-and-drop bean in session
- Names first book item
- Specifies session scope
- (1) Declares item's hover message
- Declares item's drop message
- Declares item's drag options (default)
- Declares second book item

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <managed-property>
            <property-name>dropMessage</property-name>
            <value>You are sure to enjoy.</value>
        </managed-property>
        <managed-property>
            <property-name>dragOptions</property-name>
            <value></value>
        </managed-property>
    </managed-bean>

    <managed-bean>
      <managed-bean-name>rockItem</managed-bean-name>
      <managed-bean-class>aip.ShoppingItem</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
        <managed-property>
            <property-name>hoverMessage</property-name>
            <value>Put it back.</value>
        </managed-property>
        <managed-property>
            <property-name>dropMessage</property-name>
            <value>You really don't want this one.</value>
        </managed-property>
        <managed-property>
            <property-name>dragOptions</property-name>
            <value>revert</value>
        </managed-property>
    </managed-bean>

</faces-config>
```

Declares second book item

Declares third item (rock)

Declares item's drag options (snap back)

The property named `hoverMessage` corresponds directly to the `hoverMessage` field in the `ShoppingItem` bean. JSF looks for a method called `setHoverMessage()` ❶ and calls it on the bean instance with the value we give here.

This gives us a user interface and some data, but our application can't actually do anything yet; we haven't implemented any functions that act on the user events. What does our application do? When users drag an item, the message is updated depending on what they do. As we saw in the page declaration, our JavaBean finds out that an item has been dragged through the `dragListener()` callback. The message is updated by virtue of being a text output component bound to the bean (ICEfaces ensures that it is updated in the page when necessary). Let's see how these functions are implemented in the bean (listing 9.9).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 9.9 DragDropBean

```
package aip;

import com.icesoft.faces.component           Uses ICEfaces
       .dragdrop.DragEvent;                  drag-and-drop events
import com.icesoft.faces.component.ext.HtmlPanelGroup;

public class DragDropBean  {

    private String dragMessage = "";

    public void setDragMessage(String message)  {
        this.dragMessage = message;
    }                                                      ❶ Sets/gets
                                                             dragMessage
    public String getDragMessage()  {
        return this.dragMessage;
    }

    public void dragListener(                Handles drag events
            DragEvent dragEvent)  {
        ShoppingItem item = (ShoppingItem)
          ((HtmlPanelGroup)                             Extracts shopping
                 dragEvent.getComponent())              item object
                         .getDragValue();

        if (null != item) {
          if ( dragEvent.getEventType()
                == dragEvent.HOVER_START )  {             ❷ Shows hover
            this.dragMessage =                              message if hovering
              item.getHoverMessage();
          } else if ( dragEvent.getEventType()
                == dragEvent.DROPPED )  {                 ❸ Shows drop
            this.dragMessage =                              message if dropped
              item.getDropMessage();
          }
        }

    }
}
```

The `setDragMessage()` method ❶ will likely never be called, but it's stylistically preferred to include setter methods for bean properties. If it's a hover event, we extract the hover message from the item. The item is an object that knows what should be displayed when it hovers over the shopping cart ❷. If the item has been dropped ❸, we extract the drop message from the item.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

In terms of philosophy, the main thing to notice is the loose coupling between the model and the view. A drag-and-drop event doesn't directly cause the message beside the shopping cart to be changed to a certain value; the event is received by the application and the internal state of the application is changed. The designer is free to project that internal state into the page view in any way desired, such as by displaying a message beside the shopping cart or in a pop-up, but the important thing is that the designer doesn't need to know that dragging an item to the shopping cart changes the internal state. The state change, which is in the domain of business logic, resides comfortably in the application code.

9.3 Summary

We've seen three ways to use drag and drop with Ajax: first, we learned how to apply the capability to arbitrary objects; second, we saw a refinement specifically for lists; third, we looked at the ICEfaces framework. The only changes with lists were that items automatically move out of the way and the convenient serialization function (which takes into account the position of the list items, not just which droppable they lie in). In any case, the key lesson is that JavaScript libraries (such as Script.aculo.us or ICEfaces) are available that expose drag and drop with high-level events that can be easily glued to Ajax functions. It's now just a matter of using these techniques to surprise desktop users with browser applications that are even richer than they expect.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

10 Being user-friendly

This chapter covers

- Dealing with network latency issues
- Providing context-sensitive help
- Detecting and reporting errors

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Would you rather make your users scream in frustration or purr with pleasure?

Although this may seem like a silly question (after all, who would want to intentionally frustrate their users?), it's all too easy to create user interfaces that are less than friendly. This is especially true on the Web, where our choice of controls is limited to the control set provided by the HTML specification.

Even with the limitations placed on our web applications, the same principles that are applied to their desktop brethren can be applied to our applications, despite the fact that the usability challenges faced by web applications differ from the challenges faced by desktop programs. Three such major principles that we'll examine in this chapter are

- Provide feedback—A confused user is a frustrated user. Letting the user know what's going on is one of the most important principles of usability.
- Provide proactive help—Anticipating what the user needs to know goes a long way in reducing the amount of time they sit staring at the screen, unsure of what to do.
- Be intuitive—Make sure that the user is never lost or unsure of what to do next. And most of all, never do something unexpected or unusual.

There are, of course, many other usability principles, but these are areas where web applications frequently fall flat and that we'll explore within this chapter. You'll see how using Ajax technologies can help you overcome these common stumbling blocks in order to deliver a delightful user experience.

In this chapter we'll explore these topics in the guise of the developers of a community-based recipe-sharing web application. In the spirit of the so-called "Web 2.0," this site is one in which the members of the community provide the bulk of the content, which takes the form of recipes that they share with other members. We'll just be providing the platform on which they will do it.

Because it is a community-based application, usability is a primary concern. If the members are frustrated when they attempt to use our system, contributions will suffer. And with a community-based application, contributions are what it's all about. Also, as a community-based system, it is likely to be heavily used by large numbers of members. We'll need to be sure that the user load has minimal impact on the members' experience while using our application.

So let's dig in and take a look at some of the usability challenges we face, and how, using Ajax, we can meet these challenges.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

10.1 Combating latency

Imagine a user pounding on his keyboard while shouting at the screen, "*Where's my data!*"

Delays on the Web are inevitable. Whether due to slow connections, slow servers, inefficient code, or overworked databases, sometimes our users have to wait for their data. This can be frustrating, not only because it can waste time but because often the user is left wondering, "Is anything really happening?"

This lag between the time that a request is submitted to our servers and the time that the browser receives a response with the result is known as *latency* and is one of the major usability issues for web applications.

10.1.1 Countering latency with feedback

Ajax can help reduce this latency by allowing our applications to make smaller requests and receive smaller responses, as opposed to the traditional and cumbersome full-page refresh necessary with non-Ajax web applications. While this can decrease the latency of the operations in our applications, it can also contribute to users' confusion.

How is that?

Users of web browsers are accustomed to the visual feedback cues that most browsers provide: usually a progress bar of some sort in the status bar, and a "throbber" (or other indication) in the toolbar area. Use of Ajax frequently circumvents these feedback mechanisms. So even if our Ajax applications can shorten the latency of operations, such lags cannot be eliminated completely, and our users may be left with no feedback during those delays, even if they *are* shorter.

Problem

Because we cannot completely eliminate latency in our web applications, we want to offer the user some sort of visual feedback when an operation that could be lengthy is under way.

Solution

Let's consider a page in our recipe-sharing application in which the user can search for a recipe via terms in its title. Remember that this is a community-based site, so we are anticipating (or fervently hoping) that the database for this site will contain a large number of contributions from its members. That means a search operation may take some time to complete.

Since we are going to be obtaining the search results via an Ajax request, the browser's visual cues may not be operational or sufficient to offer our users

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

proper feedback, so we need to make sure that they know that our application is working hard on their behalf. There are any number of ways we could provide such feedback, and we'll code two examples to show variations in approach: one in which a page-level loading banner is displayed, and one in which a localized animation provides a feedback cue.

Some setup details

The solutions in this chapter require the services of server-side resources that will provide the back-end activity that our application requires. Obviously a true web application back end isn't included, but only a series of servlets and classes that provide the "smoke and mirrors" to fake it enough for our purposes.

See the readme files in this chapter's downloadable source code at www.manning.com/crane2 for information on setting up the code as a Java web application that you can run. Don't worry if that sounds scary; because the downloaded code is already set up as a complete and self-contained web application, it's easier than you might think.

Because the purpose of these examples is to illustrate client-side usability techniques rather than to explore the server-side operations of a community-based site, we're not going to focus too much on the back-end operations beyond the interface provided to invoke them.

If you *are* interested in the back-end component, the source code is available in the examples for this chapter and implements a lightweight front controller employing the Command pattern. See the readme files and source code for details if you'd like.

From the point of view of the client code, all services provided by the back end are accessed via a URL of the form

```
/aip.chap10/command/WhatToDo
```

In this URL, `/aip.chap10` is the *context path* for this Java web application that was set up (if you followed the instructions in the readme files) for this chapter's code. The `/command` portion is the *servlet path* that invokes the front controller (as defined in the application's deployment descriptor), and the `/WhatToDo` is the specific *command* that we want the server to execute on our behalf.

From *our* point of view as Ajax page authors, we don't care much about the details. All we care about is that if we want to execute the `SearchForRecipes` command, the URL would be

```
/aip.chap10/command/SearchForRecipes
```

The server-side code handles the details of invoking the correct command code.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

With that behind us, let's start setting up the search page, or at least enough of it to show the point of the exercise.

Variation 1: displaying a page banner

The HTML portion of our first solution is as simple as you might expect it to be, as shown in listing 10.1. This HTML file can be found in the download for this chapter in the file `solution-10.1.1a.html` in the `solution-10.1.1` folder.

Listing 10.1 HTML for "I'm working on it" page banner

```
<html>
  <head>
    <!--script and CSS will go here-->
  </head>
  <body>
    <fieldset>
      <legend>Find Recipes</legend>          ❶ Declares fieldset and legend element pair

      <form name="searchForm"
            onsubmit="performSearch();return false;">     ❷ Defines form

        <div>                                             ❸ Specifies form controls
          <label>Where recipe title contains:</label>
          <input type="text" name="searchTerms"
                 id="searchTermsField"/>
        </div>

        <div>
          <input type="submit" value="Search!"/>
        </div>
      </form>

      <div id="resultsContainer"></div>        ❹ Creates results container

    </fieldset>

    <div id="processingNotice" style="display:none" >     ❺ Specifies initially hidden banner
      Searching! Please wait...
    </div>
  </body>
</html>
```

A `<fieldset>` and `<legend>` element pair that provides easy styling ❶ is used to hold the form ❷ containing our controls and a `<div>` ❹ in which our results will be shown.

The form controls ❸ consist of the expected combination of a label, a text field in which to enter the search terms, and a submit button.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Also notable is the `onsubmit` event handler for the form. We use the little trick of calling a function to perform our "submission operation" and then return `false` to prevent the form itself from ever being submitted to the server. Since we'll be issuing an Ajax call to the server ourselves in the `performSearch()` function, we never want the form to be submitted to its action.

Outside of the `<fieldset>`, we defined a `<div>` element containing the text "Searching! Please wait…" that is initially hidden ❺. Since we are going to apply absolute positioning to this element, it really doesn't matter where we place it in the HTML, but it is conventional to place such elements at the end of the markup, after all the relatively positioned elements.

The absolute positioning for that element, along with a few other styling rules that give it a "can't be ignored" appearance, is defined in the style header as follows:

```
#processingNotice {
  position: absolute;
  top: 0px;
  right: 0px;
  background-color: red;
  color: white;
  font-size: 1.2em;
  width: 320px;
  text-align: center;
  padding: 4px;
}
```

This not only places the element at the top right of the screen, but it also enlarges the text and gives it a hard-to-miss red background. During the search operation, the page will look as shown in figure 10.1.

The script portion of our page, defined in the `<head>` element, is shown in listing 10.2.

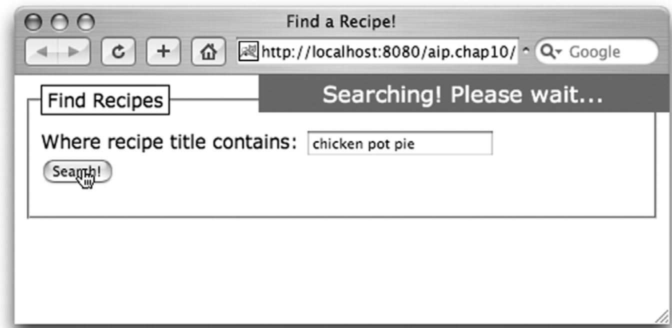

Figure 10.1
I'm working here!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 10.2 Script for page-level "I'm working on it" notice

```
<script type="text/javascript">
  function performSearch() {
    $('resultsContainer').innerHTML = '';            <-- 1 Clears out results
    new Ajax.Request(
      '/aip.chap10/command/SearchForRecipes',        <-- 2 Initiates search
      {
        onSuccess: showResults,
        onFailure: showResults,
        parameters: $('searchForm').serialize(true)
      }
    );
    Element.show('processingNotice');                <-- 3 Displays banner
  }

  function showResults(request) {                    <-- 4 Hides banner and shows results
    Element.hide('processingNotice');
    $('resultsContainer').innerHTML = request.responseText;
  }
</script>
```

The `performSearch()` function, established as the `onsubmit` event handler for our form, performs three operations. First, it clears out any previous results ❶. Our page could be used for many searches, so we remove any results that have already been displayed.

It then initiates the search by making an Ajax request ❷ to the `SearchForRecipes` command provided by our back end (as described previously) and then passing the value of the search terms field. For expediency, our smoke-and-mirrors server code completely ignores the terms and returns the results for a search on "Chicken Pot Pie" regardless of what you type into the field—hardly realistic, but good enough for our purposes.

After sending the request, the hidden banner element named `processingNotice` is displayed ❸ to let the user know that, even though nothing at all may appear to be happening, the server is busy searching for all those luscious chicken pot pie recipes.

Both the `onSuccess` and `onFailure` handlers are set to the same function: `showResults()`. This function ❹ hides the processing banner and displays the results of the search in the `<div>` element created just for that purpose. In case of an error, the error message as formatted by the server will appear in place of the results. This isn't the best possible means to display an error to the user, but it will do for now (we'll address the display of failure messages in upcoming examples).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

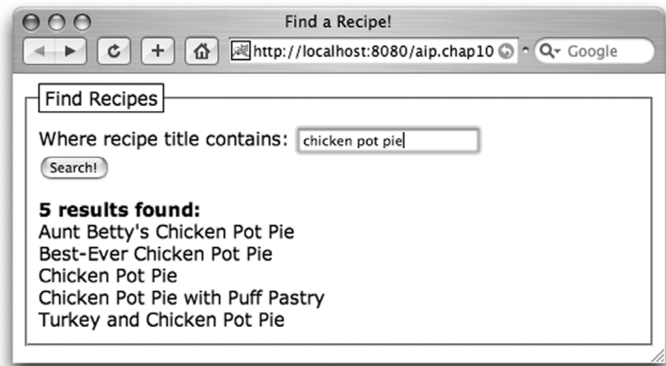

Figure 10.2
Done working!

The point is that, regardless of success or failure, the processing banner needs to be hidden upon completion of the request.

After the results have been displayed and the banner removed, the page is as shown in figure 10.2.

Obviously in the real application, we'd want to format the results a bit more nicely and probably return more information about the located recipes: perhaps ratings, submission date, submitting member, and so on. We'd also make the recipe names links to the actual recipes, but what we've got here is sufficient to make our point regarding user feedback.

The full code for this example can be found in `/solution-10.1/solution-10.1a.html` in the downloadable source code for this chapter.

It could be argued that putting a banner at the top of the page is distracting as it diverts the focus from the main elements. Let's modify our code to use a more localized variation.

Variation 2: localized animation

Flashing a banner, especially a bright red one, in the upper-right corner of the screen is sure to attract the notice of the user. That's good. We want them to notice our feedback. But there are times, especially in a longer or more complicated page, where drawing the user's eyes away from where the action is happening could be distracting to the point of interruption.

For such cases, we'd prefer that any feedback appear locally to the area in which the user is working, so let's make some changes to our original page to achieve that.

This variation requires only a few changes to the source code of our original solution, beginning with moving the `<div>` containing the processing notice. In the original recipe, we absolutely positioned that element and so we placed it at

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

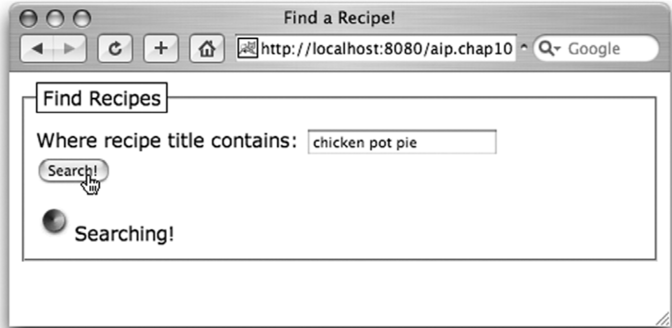

Figure 10.3 Whirling while we work

the end of the entire markup. In this variation, it's going to appear in the normal flow with other elements, so we move it to just before the results container. Now, when a search is under way, the page appears as shown in figure 10.3. The HTML file for this example is `solution-10.1.1b.html`.

This section of the markup appears in listing 10.3.

Listing 10.3 New location of the processing notice

```
      ...
      <div>
        <input type="submit" value="Search!" id="submitButton"/>
      </div>
    </form>

    <div id="processingNotice" style="display:none" >
      <img src="whirlybug.gif"/>Searching!
    </div>

    <div id="resultsContainer"></div>

  </fieldset>
```

We removed the CSS styling for this element (though note that it is still initially hidden), modified the text a bit, and added a small animation to it. The GIF is an animated icon that rotates in place, giving the illusion of activity.

That's the extent of the necessary changes.

Discussion

In this section we examined two variations on giving the user concrete feedback while a potentially lengthy Ajax operation is under way. The use of Ajax reduces

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the latency of this operation as only the results need to be fetched from the server (as opposed to the markup for the entire page), but because we anticipate a large database and heavy load, the operation could still take a nontrivial amount of time.

In the server code for the search command, the operation is artificially extended to take 5 seconds—though on first run it might take longer as the servlet engine gets up on its feet. That way, we may see the effects of the feedback display.

In one variation we used a hard-to-ignore, page-level banner, while our second variation used an in-place animation. In either case, the user is made aware that something is happening on their behalf, which should help keep them from wondering if something is really going on. It should also help prevent repeated button presses by impatient or bewildered users that might overload our already busy database.

One caveat: be careful with animations, as it's all too easy to go overboard with them. Something subtle like our whirling ball, or a hypnotic barber pole, is quite sufficient. You don't need a hand punching through the page, a construction worker digging away, or any of the other overused and rather obnoxious animations that are prevalent on the Web. As long as you remember the "keep it subtle" rule, something thematic for the site might be appropriate—perhaps a spinning fork and spoon for our recipe site? But we suggest you avoid a dancing banana.

10.1.2 Showing progress

The variations on the "I'm busy" notice that we explored in the previous section are all well and good for letting the user know that something is happening in the background. But while they give the user a warm and fuzzy feeling that *something* is going on, they do not give the user any indication of *where* in the process that operation is at any time.

Much of the time, it is quite difficult to predict how long a back-end operation might take. But when such information *is* possible to obtain, a *progress bar* is a good means to give the user feedback regarding the expected completion time of an operation.

Problem

We'd like to give users more feedback than just "things are happening." We want to give them an indication of *where* they are in the progress of the operation.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Solution

In this section we'll create a progress bar that can indicate what percentage of a server-side operation is completed. We'll consider only the client side of the equation here. Instrumenting server-side operations to be able to report this type of information is well beyond the scope of this section; in fact, it could consume an entire chapter of a book on server-side threading. We'll simply assume that the server is capable of giving us that information whenever we ask for it.

The previous solution just included the code and elements necessary to display our processing notification directly in the page markup and code. Since the progress bar is a little more involved, we'll abstract it out of the page into a reusable JavaScript class.

Let's get right to it.

The ProgressBar class

Factoring the code for the progress bar into its own JavaScript class not only simplifies the page that it will appear on, but ensures that it can be easily used on any page, or even multiple times on a single page.

Because we will be using the Prototype pattern for creating classes, we start our `ProgressBar.js` file with the usual declaration:

```
ProgressBar = Class.create();
```

The rest of the declarations are made in the class's `prototype` property, including the initializer for the constructor, which is defined in listing 10.4.

Listing 10.4 Initializer for the ProgressBar class

```
initialize: function(parent,options) {
  this.parentElement = $(parent);
  this.options = Object.extend(
    {
      className: 'progressBar',
      color: 'red',
      interval: 2500
    },
    options
  );
  this.parentElement.innerHTML = '';
  this.parentElement.style.display = 'none';
  this.barContainer = document.createElement('div');
  this.barContainer.className = this.options.className;
  this.barContainer.style.position = 'relative';
  this.bar = document.createElement('div');
  this.bar.style.position = 'absolute';
  this.bar.style.height = '16px';
```

1. Declares initializer with constructor signature
2. Merges passed options with defaults
3. Ensures empty parent element
4. Creates progress bar elements

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    this.bar.style.width = '0%';
    this.bar.style.backgroundColor = this.options.color;
    this.barContainer.appendChild(this.bar);
    this.parentElement.appendChild(this.barContainer);
  },
```

The signature for this method ❶ specifies two parameters: one that identifies the parent element within which the progress bar will be placed, and a hash of options. In typical Prototype fashion, the parent element can be identified by reference or by `id`.

The options passed by the user are merged with a set of defaults that we provide ❷. The options are

- `className`, the style class name to be applied to the progress bar. If omitted, a default of `progressBar` is used.
- `color`, the color to be used to fill in the progress bar as the operation progresses. The default is red.
- `interval`, the interval at which the server should be polled for updated completion percentage. A default of 2.5 seconds is supplied.

We make sure that the parent element is empty ❸ and hidden, and then proceed to create the progress bar elements within it ❹. Note that we used the `innerHTML` property to empty the parent (because it's easy) and the DOM API to create the elements (to avoid building markup in strings).

The DOM manipulation that we apply here creates the equivalent of the following markup:

```
<div class="nameFromOptions" style="position:relative;">
  <div style="position:absolute;height:16px;width:0%;
              background-color:fromOptions;">
  </div>
</div>
```

The outer `<div>` serves as the progress bar itself and its appearance is controlled by the CSS class name in the options set. The inner `<div>` is the sliding bar filled with a color from the options that indicates the percentage of the operation that is complete. Its width is initially set to 0%.

Once a page creates an instance of the progress bar, we're all set to start it. For that, we provide a `start()` method. Of course, if we allow the page code to start the progress bar, we also need to be able to stop it. Both methods are shown in listing 10.5.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 10.5 Starting and stopping the ProgressBar

```
start: function() {                          <-- 1 Starts progress bar
  this.bar.style.width = '0%';
  Element.show(this.parentElement);
  this.timer =
    setInterval(this._tick.bind(this),this.options.interval);
},

stop: function() {                           <-- 2 Stops progress bar
  clearInterval(this.timer);
  Element.hide(this.parentElement);
},
```

In the `start()` method, we set the percentage `<div>` to a width of 0% just in case this is not the first time that the progress bar has been started ❶, and reveal the progress bar by showing its container, which we made sure was hidden by the constructor.

We then start a timer to begin polling the server for the completion percentage of the ongoing operation using the `setInterval()` function. We specify an internal implementation method as the callback, and set the duration of the interval at which it will be invoked to whatever value ended up in the options hash.

We store the handle to the timer in an instance member named `timer`. We need to store this handler since the only way we will be able to later stop the timer is by using the handle. Also note that we bind the callback to the object instance. If we failed to do this, the context object for the callback would be the window rather than our instance of ProgressBar.

The `stop()` method cancels the timer by calling `clearInterval()` on the stored timer handle and hides the progress bar ❷.

Once the timer is established by `start()`, the `tick()` callback will be invoked regularly at the specified interval until we stop it. It will be the duty of that callback to ask the server for the completion status and to adjust the progress bar display as appropriate. This method is shown in listing 10.6.

Listing 10.6 Tick, tick, tick

```
_tick: function() {
  var self = this;
  new Ajax.Request(
    '/aip.chap10/command/GetProgress',
    {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        onSuccess: function(request) {
          self.bar.style.width = request.responseText + '%';
        }
      }
    );
  }
```

Each tick of the interval timer created in the `start()` method results in an invocation of the `_tick()` method. The leading underscore in the method name is a convention that identifies this method as an implementation method for the class's own internal use and that should not be called by external code.

The method stores a reference to the instance in a variable named `self` since we'll want to be able to refer to the instance in a closure that will be created later in the method. Remember that closures do not include `this`.

After that, the method makes an Ajax request back to the `GetProgress` command in order to ask the server for the percentage of completion of an operation. In an actual implementation, the server would probably require a job ID or other identification of the operation to be measured, but since we're just faking it for the purposes of this example, we don't make any assumptions as to what a real-world server would require.

The success handler for this request is a closure that obtains the percentage of completion from the response and adjusts the width of the progress bar's inner `<div>` accordingly.

That's it for the ProgressBar implementation; let's see how we can use it.

Using the ProgressBar on a page

Let's adjust the code of the second variation of section 10.1.1 to use an instance of ProgressBar in place of the simple animation. Because we abstracted the code to create and manipulate the progress bar into its own class, the modifications are gratifyingly simple.

Once we've finished, when displayed and the search operation is under way, the page appears as shown in figure 10.4.

Of course, we first need to import the class:

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

Then, in the `performSearch()` function, after the request has been created, we replace the code to show the processing notice with the following code to create and start an instance of ProgressBar.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

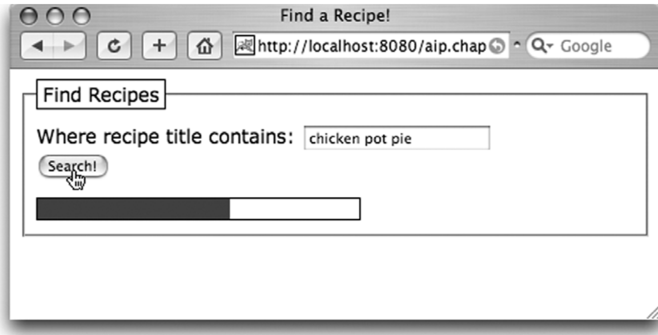

Figure 10.4
In progress!

```
if (!window.progressBar) {
  window.progressBar = new ProgressBar(
    'progressBarContainer',
    {
      className: 'progressBar',
      color: 'blue',
      interval: 1000
    }
  );
}
window.progressBar.start();
```

This code fragment checks to see if we've created a progress bar yet, and if not, instantiates one that will poll every second and use blue as its color. Once we're sure a progress bar exists, it is started.

The `showResults()` function is modified to stop the progress bar:

```
function showResults(request) {
  window.progressBar.stop();
  $('resultsContainer').innerHTML = request.responseText;
}
```

In the body markup we replace the processing notice with the container to hold the progress bar:

```
...

<div id="progressBarContainer"></div>

<div id="resultsContainer"></div>
```

Finally, we create a style class for the progress bar:

```
<style type="text/css">
  .progressBar {
    position: relative;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      width: 256px;
      height: 16px;
      border: 1px solid black;
    }
  </style>
```

The complete code for this solution can be found in the files `solution-10.1.2.html` and `ProgressBar.js` in the `solution-10.1.2/` folder in the downloadable code for this chapter.

Discussion

In this section we've provided a means to give the user more precise feedback by letting them know exactly where they stand in a long-running process. This is superior to the static feedback we explored in the previous section *if* the percentage of completion is available from the server and *if* it is accurate.

If the information you are going to get from the server doesn't reasonably reflect the actual situation, it's better to stay with *less* information than *incorrect* information. In some cases, the completion percentage may be a best guess but still close enough to be useful. In these situations, it's always better to underestimate the portion that is complete rather than to overestimate.

Users will be delighted when a process ends earlier than expected, but will just think that our application is stupid if it stays pegged at 100% before the processing completes.

Our ProgressBar class is quite usable as is, but could stand some improvement; you might try your hand at these:

- Show the percentage amount in the middle of the progress bar.
- Allow the inclusion of a label or other text.
- Give the page authors more control over the styling of the inner `<div>`.

The solutions we've examined thus far have been focused on letting the user know what's happening *now*. Let's take a look at some ways to help the user by *anticipating* their needs.

10.1.3 Timing out Ajax requests

Left to its own devices, an XHR request will not time out. But it's a real possibility that network latency may cause requests to take a long time to respond. Rather than just letting the request languish, what can we do to detect and stop runaway requests?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

We want to stop an Ajax request that's taking longer than a duration we'll specify.

Solution

XHR makes no provisions for timing out after a specified period of time. But it does have an `abort()` method that we can call to stop a request dead in its tracks if we can just figure out a way to call it at the appropriate moment.

The JavaScript window object has two methods that could help us here: `window.setTimeout()` and `window.setInterval()`. Although these functions are fairly similar, the first establishes a one-shot timer, while the latter establishes a repeating timer. As we only wish to stop our request once at the end of a specified timeout period, we'll use the `setTimeout()` method.

Our approach will be to start a timer just after we initiate a request, and if the timer expires before the request completes, we cancel the request.

That sounds fairly straightforward, but there are a few nuances to consider with regard to making sure that we have access to the information that we'll need *when* we need it.

Let's revisit the solution in section 10.1.1. In the second variation on that solution we displayed a little whirling ball and message while a long request was processing on the server. Let's explore how to modify that code to time out if the request does not complete in a set period of time—let's say 3 seconds.

To accomplish this we'll start a timer immediately after the request is generated. That's easy to do with a call to `setTimeout()`, but we'll also need to make sure that we can cancel this timer if the request completes before the timeout period. We therefore need to store the handle returned by `setTimeout()` somewhere so that we can access it in the request's success handler.

And we want to do so *without* resorting to global variables.

The code in listing 10.7 shows the reworked `performSearch()`and `showResults()` functions from that previous example, with changes and additions highlighted in bold. The modified HTML page can be found in the file `/solution-10.1.3/solution-10.1.3.html`.

Listing 10.7 Reworked "working" example

```
function performSearch() {
  $('resultsContainer').innerHTML = '';
  var request = new Ajax.Request(            <-- 1 Creates and records request reference
    '/aip.chap10/command/SearchForRecipes',
    {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      onSuccess: function() { showResults(request); },     ❷ Uses closures
      onFailure: function() { showResults(request); },
      parameters:
        $H({
          terms: $F('searchTermsField')
        }).toQueryString()
    }
  );                                                  ❸ Sets timer
  request.timer = setTimeout(
    function() {
      request.transport.abort();
      Element.hide('processingNotice');
      $('resultsContainer').innerHTML = 'Request timed out!'
    },
    3000
  );
  Element.show('processingNotice');
}
                                                      ❹ References request
function showResults(request) {
  clearTimeout(request.timer);
  Element.hide('processingNotice');                    ❺ Cancels timer
  $('resultsContainer').innerHTML =
    request.transport.responseText;
}
```

We've made four changes to the code in order to institute the timeout timer. The major addition to the code is to add a call to `setTimeout()` immediately after we dispatch the Ajax request to the server ❸, and store the timer handle returned from the call in a property named `timer` on the request object. This eliminates the need for a global variable in which to store the timer handle, but it also means that we need to add a local variable ❶ that stores the created Ajax.Request instance. Since local variables create none of the problems that global variables introduce, this is a more-than-acceptable trade-off.

The callback function for the timer uses the request reference to locate the transport instance (the actual XHR object), and calls its `abort()` method to cancel the request. It then hides the processing notice and informs the user that the request has timed out by placing a message to that effect in the results area.

That takes care of the situation in which the server process will last longer than the acceptable timeout value. But what about the case where the process does not take too long? When the process completes before the timer expires, we need to cancel the timer lest it fire its callback and incorrectly inform the user that the process timed out when it actually did not.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The problem we have in this case is that the `onSuccess` handler is passed a reference to the XHR instance but has no access to the Ajax.Request instance that is holding the timer handle. And we need that to cancel the timer.

Our first instinct might be to store the timer handle on the transport (XHR) instance rather than on the Ajax.Request instance. If we could do that, we'd have access to the timer handle in both locations that we need it: the timer callback *and* the `onSuccess` handler. Indeed, that approach would work well in browsers such as Safari and Firefox where XHR is implemented as a native JavaScript object. But IE 6 implements XHR as an ActiveX object, upon which we are not allowed to create properties.

Pooh!

Since we're stuck tacking the timer handle onto the Ajax.Request instance, we need to find a way to make a reference to that instance available to the `onSuccess` handler.

To do so, we modify the code of the original solution, replacing direct references to the `showResult()` function as shown:

```
onSuccess: showResults,
onFailure: showResults,
```

with inline functions ❷ that include the closure containing the request reference so that we can pass the request instance to the `onSuccess` handler:

```
onSuccess: function() { showResults(request); },
onFailure: function() { showResults(request); },
```

This means that the Ajax.Request instance is what is passed to the `showResults()` function in place of the XHR instance. This allows us to access the stored timer handler as well as the XHR instance in that function ❹.

Now, in that function, we cancel the timer ❺ and adjust the reference to the transport when obtaining the response text.

Discussion

With this solution, we now have the ability to cancel an Ajax request that's taking longer than we wish to allow for it. And we've done so without messing up the page with global variables.

The code for this example was lightweight enough that we didn't abstract it out to its own class as we did with the progress bar code. Small footprint that it may have, it's still more code than we'd be comfortable simply copying from page to page.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We've already taken a small step to integrating this functionality with that of Ajax.Request by piggy-backing the timer handle on an instance of that class. So an interesting approach to organizing this code could be to *fully* integrate it into a request class.

Using the knowledge that we gained in chapter 3 on using the Prototype mechanism to extend classes, can you figure out a way to extend Ajax.Request into a new class that incorporates the timeout feature?

Imagine a class, perhaps named Ajax.TimedRequest, that we could call as follows:

```
new Ajax.TimedRequest(
  '/aip.chap10/command/SearchForRecipes',
  {
    onSuccess: showResults,
    onFailure: showResults,
    onTimeout: reportTimeout,
    timeoutPeriod: 3000,
    parameters:
      $H({
        terms: $F('searchTermsField')
      }).toQueryString()
  }
);
```

How cool would that be? Why not give it a try?

Now we'll turn our attention to another problem associated with long latencies: the impatient user.

10.1.4 Dealing with multiple clicks

Nothing can kill a server faster than hammering it with too many requests at a time. This is always a concern to us as Ajax developers; even though we are making smaller requests, we're probably making many more of them than in a traditional web application.

We've seen some solutions in this chapter that try to dissuade users from getting impatient and submitting multiple needless requests by keeping them informed and aware that processing is occurring on their behalf. But even so, you *know* there are going to be users out there who, though informed and aware, are going to be clicking away regardless, screaming "Hurry up!" at the screen.

And it's not only the impatient user that we need to contend with. Users are also conditioned to double-click on items in desktop applications in order to activate them. Granted, most buttons in such applications are invoked using a single click, just like in our web applications, but some habits die hard and

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

there is going to be a percentage of our user base that will mercilessly double-click our buttons.

Problem

Users can be impatient, absent-minded, clumsy, or all three. We want a means of preventing multiple requests for the same operation from being submitted while that operation is already under way.

Solution

The good news is that we can use a fairly simple technique to nip these users in the bud: we'll just disable the source of the clicks—namely, the button—whenever we're already busy processing a first click on it.

Let's once again start with the "whirling ball" solution from section 10.1.1 and enhance it to prevent multiple clicks. Figure 10.5 shows the finished page while a search is under way.

Figure 10.5 Try to double-click! Just try!

The code shown in listing 10.8 (and available in the file `/solution-10.6/solution-10.6.html`) shows the modified fragments of the original solution with the changes and additions highlighted in bold.

Listing 10.8 Stop the insanity!

```
function performSearch(button) {
  button.disabled = true;
  $('resultsContainer').innerHTML = '';
  var request = new Ajax.Request(
    '/aip.chap10/command/SearchForRecipes',
    {
      onSuccess: function() { showResults(request); },
      onFailure: function() { showResults(request); },
```

(1) Disables button
(2) Uses closures to pass instance

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      parameters: $('searchForm').serialize(true)
      trigger: button          ◁─ ❸ Stuffs button into options
    }
  );
  Element.show('processingNotice');
}

function showResults(request) {
  request.options.trigger.disabled = false;     ◁─ ❹ Enables button
  Element.hide('processingNotice');
  $('resultsContainer').innerHTML =
    request.transport.responseText;
}

...

<form id="searchForm" onsubmit="return false">     ◁─ ❺ Disables form submission

...

<input type="button" value="Search!"        ❻ Passes button to handler
       onclick="performSearch(this) "/>
```

The changes actually start near the bottom of the page where we make a change to the button element itself ❻. Rather than use an input type of `submit`, we use a type of `button`, which prevents the button from triggering a form submission. We use an `onclick` event handler to trigger the `performSearch()` function, passing a reference to the button.

This is a technique that we could have applied to previous solutions, but it becomes essential now that we've added an `onclick` event handler to the button. Otherwise, a click of the button could create a race condition where an `onclick` event handler *and* a form submission are both initiated.

Because the handler will now be triggered directly by the button, the `onsubmit` event handler of the form ❺ is modified to simply return `false`, which prevents accidental form submission (this might occur if the user hits the Enter key while in a text field).

The `performSearch()` function, which is now passed a reference to the triggering button, immediately uses that reference to disable the button ❶ before our impatient user gets a chance to click it again.

Were this a traditional web application, that'd be all we need to do. The page refresh that would occur when the response was returned would take care of reenabling the button. But this is an Ajax application and we're not going to be

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

refreshing the page, so we need to make sure that once it's OK to initiate the search again we reenable the button.

In the `onSuccess` handler ❷ for the request, we'd like to reenable the button, but we don't have a reference to it. And we don't want to hard-code it; there may be more than one button that could have initiated the action.

If you remember our most recent solution, we had a similar need to store a timer handle, so we tacked it onto the Ajax.Request instance itself and used closures to make sure that instance was available to the `onSuccess` handler. We'll use a similar scheme here, but with a slightly more clever twist.

In the timeout solution, we added the timer handle to the request instance *after* the instance had been created. In this solution, we'll record the reference to the button *in* the options hash that we pass to Ajax.Request, in a property named `trigger` ❸.

What's that? How can we get away with such shenanigans? Ajax.Request doesn't know beans about any option named `trigger`.

The Ajax.Request class handles its options in much the same manner that we have handled option hashes in the classes we have created in this chapter: by merging the passed object with an internal object containing the default values. When this merge is complete, the resulting object contains the union of all properties in both objects. So any properties that we place on the hash passed into Ajax.Request will end up in the final options object and get carried along for the ride.

Since Ajax.Request doesn't know anything about these extra options, it will never try to access them and won't even know that they're there. But we'll be able to reference them wherever we have access to the request instance.

So when the `onSuccess` handler `showResults()` is invoked with the request reference passed as a parameter, we can use the button reference that we stashed into the options hash to reenable the button ❹.

Look, Ma! No global variables. No hard-coded assumptions.

Discussion

Now that we have figured out a way to prevent multiple clicks that initiate the same operation while one is already running, our server can rest easy—can't it?

Well, no. We still need to make sure that the server code can handle multiple requests, especially in cases where such a repeated request can have catastrophic results, such as database operations that delete or add records.

Remember, even though our users may be impatient or clumsy, they can also be clever and malicious. Server-side code must always be on its guard. So while

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

a technique such as the one described in this section may not let us be lazy when writing the server code, it will help us keep network traffic down for accidental cases.

We also explored a clever means of avoiding global variables or hard-coded references in our handlers by using the options hash passed to Ajax.Request to carry the reference to the triggering button. As an exercise, you may want to revisit the timeout solution and retrofit that example to use this technique.

10.2 Preventing and detecting entry errors

Another source of user frustration concerns data entry errors.

Users *are* going to make mistakes when they enter data. The mistakes may stem from anything, from clumsy typing to genuine confusion over what is expected. In this section we'll explore some solutions for dealing with entry errors.

We'll begin by looking at a way that we can anticipate the users' needs and ensure that they understand what is expected *before* they type anything in. Then we'll see how to let them know, in as helpful a way as possible, when entry errors do occur.

10.2.1 Displaying proactive contextual help

Most help systems display their information *after* the user asks for help on a control or subject; that is, if the user even *knows* that help is available. Some applications bury their help so deep that the users never even realize that it is there.

Other applications may have help systems that do not provide contextual help. Instead of giving help on the task the user is currently occupied with, a huge list of available help topics is displayed, leaving the user to hunt around and guess which topic might contain useful information.

In this solution we'll look at a way to retrieve and display help associated with the controls of a form in a proactive *and* contextual manner.

Problem

We want to retrieve and display help text associated with the controls of a form as each control is visited. The help should be proactive and should not interfere with the user's workflow when filling in the form's data.

Solution

As we did for the previous solution, rather than putting the code for our help system on the page, we'll factor it out into a JavaScript class. That way, not only will we keep the level of complexity of the page code down, but we'll also have a reusable component that we can use on many pages.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Our class, which we'll name HelpConveyer, will find all the elements of a passed form and instrument them to contact the server for related help text whenever the element gains focus.

The HelpConveyer class

As usual, we declare our JavaScript class using the Prototype mechanism as

```
HelpConveyer = Class.create();
```

The initializer, which will turn out to be the only method in our class, is defined in the `prototype` property, as shown in listing 10.9.

Listing 10.9 Initializer for the HelpConveyer class

```
initialize: function(targetElement,form,url,paramName) {      ❶ Accepts initialization parameters
  this.target = targetElement;
  this.form = form;
  this.url = url;
  this.paramName = paramName;
  var conveyer = this;
  $A(this.form.elements).each(                    ❷ Iterates over form elements
    function(control) {
      if (control.name != undefined && control.name != '') {
        control.onfocus = function() {                          ❸ Assigns onfocus handler
          var paramHash = {};
          paramHash[conveyer.paramName] = control.name;
          new Ajax.Updater(                       ❹ Fetches help text
            conveyer.target,
            conveyer.url,
            {
              method: 'get',
              parameters: paramHash
            }
          );
        };
      }
    }
  );
}
```

This constructor-invoked initializer accepts four required parameters ❶: a target element in which help text will be displayed; a reference to the form whose elements are to be instrumented; the URL to invoke when asking for help text; and the parameter for that URL with which to pass the help topic to be retrieved. For our purposes, the name of the field will be used as the help topic.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

After storing the parameter values in instance members, a local variable named `conveyer` is created, which is assigned a reference to the current instance. By now you should recognize this as a sign that closures will be employed later in the method.

Each element in the passed form is iterated over, using Prototype's `each()` method ❷. If the element has a `name` attribute that is not blank, a function (and its closure) is assigned as the `onfocus` event handler for the element ❸.

This inline event handler function creates an object named `paramHash` to contain the topic parameter that will be converted to a query string during the creation of the Ajax request.

Then the services of Prototype's Ajax.Updater ❹ is employed to fetch the help text from the server and stuff it into the passed target element.

That's a big payback for the work of one little initializer. Each element in the form now has the ability to fetch and display its own help text. Let's see how we'd use this class on our pages.

Using the HelpConveyer class

Because we abstracted all the hard work into the HelpConveyer class, using the class on our pages is a simple matter of instantiating the class with the appropriate parameters.

Let's imagine the page in our recipe-sharing application in which our members will enter or edit recipes. As you might imagine, there are more than just a few fields that need to be filled out. We'll initially design the form with the following fields:

- Title
- Source
- Minimum number of servings
- Maximum number of servings
- Ingredients list
- Steps

These last two fields are interesting in that they will be implemented using text areas. For the list of ingredients, each ingredient should be entered one per line in the text area. For the steps, each should be entered in the text area, with a blank line separating the individual steps. When displayed, our page looks as depicted in figure 10.6.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

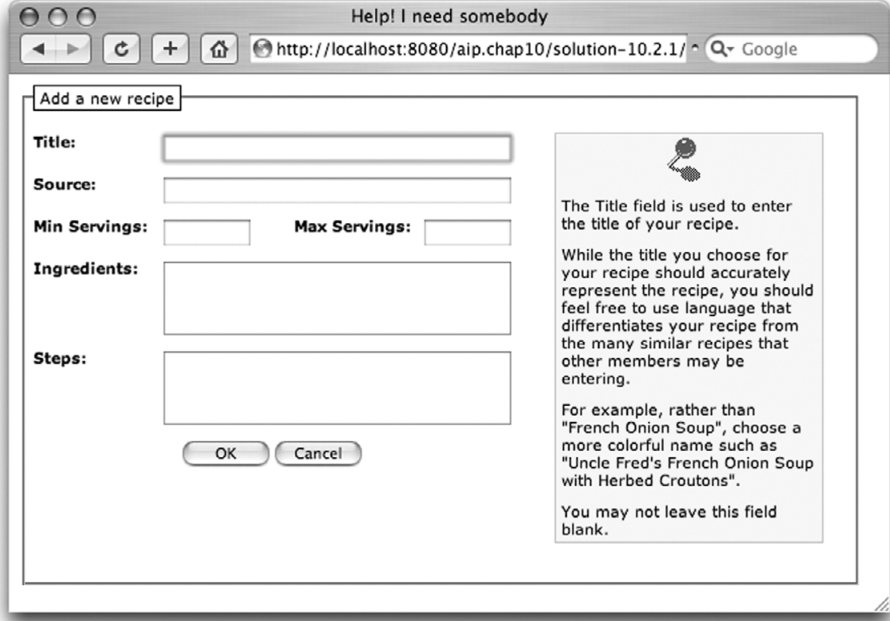

Figure 10.6 The help-enabled recipe form

None of this will be obvious to the members, so our proactive help system should prove quite useful for this form, as shown in listing 10.10.

Listing 10.10 The recipe form

```
<div id="formControls">
  <form name="recipeForm">
    <div class="line">
      <label>Title:</label>
      <input type="text" name="title" id="titleField"/>
    </div>

    <div class="line">
      <label>Source:</label>
      <input type="text" name="source" id="sourceField"/>
    </div>

    <div class="line">
      <label>Min Servings:</label>
      <input type="text" name="minServings"
             id="minServingsField" />
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        <label id="maxServingsLabel" >Max Servings:</label>
        <input type="text" name="maxServings"
               id="maxServingsField"/>
      </div>

      <div class="line">
        <label>Ingredients:</label>
        <textarea name="ingredients" id="ingredientsField"
                  rows="4"></textarea>
      </div>

      <div class="line">
        <label>Steps:</label>
        <textarea name="steps" id="stepsField"
                  rows="4"></textarea>
      </div>

      <div align="center" id="buttonBar">
        <input type="button" value="OK" name="okButton"/>
        <input type="button" value="Cancel" name="cancelButton"/>
      </div>

    </form>
  </div>
```

Each line of the form is enclosed in a <div> element that contains a <label> and an <input> element; in one case, two elements with their labels ganged up into a single line. CSS rules (defined in the page header) are applied to lay the form out in a pleasing manner.

We also need someplace to show the help text that is to be associated with each field. We define that area as a series of nested <div> elements, as shown in listing 10.11, with the innermost <div> being the one in which help text will be inserted. The reason for the nesting is so that we can apply some interesting styling to this construct.

Listing 10.11 The help text display area

```
<div id="helpContainer">
  <div id="helpSticky">
    <div id="helpDisplay">
    </div>
  </div>
</div>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The form and this help container are placed side by side on the page—form on the left, help on the right—so that the user can easily see the help text while working on the form. We also want the help text area to stand out from the form, so we'll apply some clever styling and use a simple GIF image in order to make it look like a yellow "sticky note" tacked to the page with a pushpin. (Why you'd need to use a pushpin to keep a sticky note in place is beyond me, but it *does* look cool.)

We won't elaborate on the details of the CSS to accomplish this, but you can find it in the downloadable source code for this chapter in the file `/solution-10.3/solution-10.3.html`.

Note how the help text associated with the Title field has already been displayed as a result of that field gaining focus at page load. Let's see how that was accomplished. The `onload` event handler for this page is defined in listing 10.12.

Listing 10.12 Loading up help in onload

```
window.onload = function() {
  new HelpConveyer(                          <-- 1 Creates instance of HelpConveyer
    'helpDisplay',
    document.recipeForm,
    '/aip.chap10/command/GetHelp',
    'topic'
  );
  document.recipeForm.title.focus();         <-- 2 Assigns focus to first field
}
```

As it turns out, we didn't have to do much work at all. We create an instance of the HelpConveyer ❶, identifying the target area that is to display the help text, the recipe form, a URL that will return the help text, and the parameter expected by the resource at that URL to pass the help topic.

Then we assign focus to the first field (Title) in the form ❷. The code we set up in the HelpConveyer class does all the rest!

As we tab through the various fields in the form, the help text automatically updates with the help text specific to that field. Figure 10.7 shows the page after tabbing down to the Ingredients field.

What could be more user-friendly?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

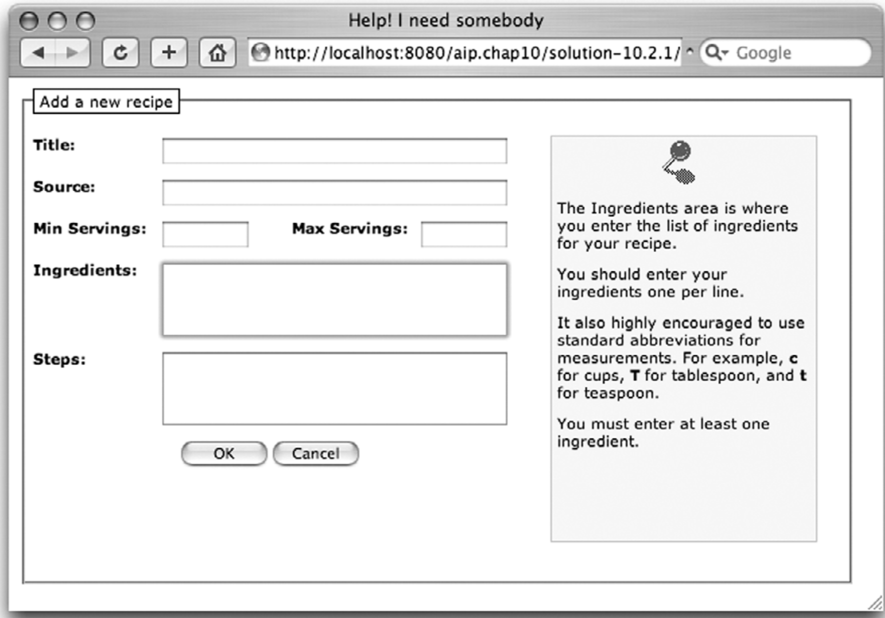

Figure 10.7 Entering ingredients

Discussion

In this section we've set up a technique to display context-sensitive help to the user in a proactive manner.

The user doesn't need to go through the motion of asking for help, or even figure out if help is available and how to get it. This technique is also extremely unobtrusive, as users do not have to interrupt their text entry workflow in order to receive help.

For this solution, we used a command URL that accepted the help topic (consisting of the name of the field) as a parameter. This could easily have been set up to just look for an HTML file with the topic's name in the local folder.

Useful as what we have set up here may be, there's always room for improvement.

After entering four or five recipes, our members will probably be old hands at using our form, and the help text may become more distracting than helpful. We might want to consider implementing some means of letting members turn it off once they become experts at using the system.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We've also made the simple assumption that each help topic can be uniquely identified by the name of the field. In anything but the most trivial of applications, this is likely to lead to naming collisions. Beefing up the manner in which topic keys are determined would be an important improvement. Perhaps the combination of form name and field name would suffice?

By providing proactive help, we've reduced the number of errors that our members will make when filling in the form, but we can't expect that mistakes will never be made. Let's take a look at checking for them.

10.2.2 Validating form entries

While it is never a good idea to assume that our users are stupid, it's an equally bad idea to assume that they're not going to make mistakes. Whether it's because they don't understand what they are supposed to do (the contextual help mechanism of the previous section should help minimize this factor), are distracted, are being careless, or are just suffering from "fat fingers syndrome," invalid entries are going to be made in our form fields.

In a traditional web application, form entry data validation can take place on the client in a limited fashion, and should *always* take place on the server (regardless of whether client-side validation was implemented).

Client-side validation in such applications can only perform simple checks that don't need any context or information that is only available on the server. For example, simple JavaScript can be used to verify that required fields aren't left blank or that numeric fields contain valid values. But validation checks such as checking zip codes against addresses, checking the validity of credit card numbers, or any other check that requires more information than can be made available to the client must be performed on the server after the form has been submitted.

As long as the user is returned to the form without losing any data, and with the errors clearly identified, this is not a disaster. But it's not as friendly as it could be, either.

Wouldn't it be better to inform the user of entry errors immediately? With Ajax, we have the ability to perform server-side checks without having to wait for the form submission.

Problem

We want to perform field-by-field server-assisted data validation on form fields. We want our solution to be reusable, and we want validation failure messages to be presented to the user in a consistent manner regardless of whether a check is occurring on the client or server.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We touched upon form field validation in chapter 5 and again in chapter 6. This section presents yet another take on this very important subject.

Solution

In the same way that we used the `onfocus` event handler of form fields to display proactive help to our users in the previous solution, we can use the `onblur` event handler to detect when a user is leaving a field. In cases where the field requires validation that can only be performed on the server, we'll set up a mechanism to handle the validation and report errors.

It'd be easy to just write handler functions that use Ajax on a field-by-field basis, but we're smarter than that. We want our solution to be reusable, and we want to make sure that we're not just cutting and pasting the same code over and over again on the pages.

So let's dig in, but better hold on to something. This is going to be one of the more complicated solutions in this chapter. And when we're done, we still won't have as robust a validator framework as we'd eventually like to have, but we *will* have a solid foundation on which to build that framework.

Strap in, and let's go.

The FieldValidator class

We'll define a class named FieldValidator that is going to meet a rather strident set of requirements:

- An instance of a FieldValidator will be created for each field that requires validation, and will handle validation errors in a consistent manner.
- The validator will operate via plug-in "verifiers" that can be defined as either a client- or server-side operation. Users will be unaware of where the validation takes place.
- A handful of common client-side verifiers will be provided. Page authors will be able to plug in custom verifiers.
- Upon a validation error, the appearance of the field will be altered and a validation failure message will be presented to the user.
- When a field is corrected and no longer fails validation, the original field appearance will be restored and the failure message removed.
- All of this will occur in real time triggered as the user leaves the field.

Whew! That's no small laundry list. Let's start by defining the constructor, as usual, with

```
FieldValidator = Class.create();
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The initializer for the class is shown in listing 10.13.

Listing 10.13 Initializer for FieldValidator

```
initialize: function(field,verifier,options) {      ❶ Declares initializer signature
  this.field = $(field);
  this.verifier = verifier;
  this.options = Object.extend(                     ❷ Merges options with defaults
    {
      errorContainer: 'errorContainer',
      errorClassName: 'fieldInError',
      paramName: 'value'
    },
    options
  );
  this.errorContainer = $(this.options.errorContainer);   ❸ Refers to error message container
  this.field.validator = this;                       ❹ Refers to field's validator
  this.field.onblur = function() {                   ❺ Validates on loss of focus
    this.validator.validate();
  }
},
```

This initializer accepts three parameters ❶: the ID or reference to the element to be instrumented with the validator, the verifier to be used to validate the field's data, and a hash of options.

The *verifier* can be either a JavaScript function that will perform client-side validation, or the URL to a server-side resource to perform the validation. If a JavaScript function is provided, it will be called at the appropriate time with a parameter consisting of a reference to the instrumented field. It is expected that this function will return `null` if the field passes validation, or if validation fails, a string containing a message explaining the failure.

The FieldValidator class includes two built-in client-side verifiers providing validation checks that are frequently employed. We'll talk about those later after we're done discussing the initializer method.

If the URL of a server-side validation resource is specified as the verifier, an Ajax request to the resource is initiated at validation time. The value of the field to be validated is passed to the resource using a request parameter whose name is provided as an option in the options hash.

After storing the field reference and verifier in instance members, any options passed in by the page author are merged with default values ❷. The supported options are as follows:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- `errorContainer`, the ID or reference to a container where validation failure messages are to be displayed
- `errorClassName`, the style class name to be applied to fields that fail validation
- `paramName`, the request parameter name used to pass the field value to server-side validation resources

Once the error message container has been determined, a reference to it is stored as a member ❸ for easy reference later.

A reference to the field's validator is added to the field element itself ❹, which is used in the function assigned as the `onblur` event handler for the field ❺ in order to call the `validate()` method when the user tabs out of the field. After initialization is complete, the field is ready for validation whenever the `onblur` event handler is triggered.

Earlier we mentioned built-in verifiers. The FieldValidator class provides two, as shown in listing 10.14.

Listing 10.14 The built-in verifiers

```
FieldValidator.verifier.NotBlank = function(field) {
  if ($F(field) == '') {
    return 'The ' + field.name + ' field cannot be blank';
  }
  else {
    return null;
  }
}

FieldValidator.verifier.IsNumeric = function(field) {
  if ($F(field) == '' || isNaN(new Number($F(field)))) {
    return 'The ' + field.name + ' field must be numeric';
  }
  else {
    return null;
  }
}
```

The first verifier reports a validation failure if the field's value is blank, while the second ensures that the value, if not blank, is numeric.

Note that these functions are not declared as part of the class's `prototype` property. Referencing them does not require an instance of FieldValidator, as shown in this example setup of a validator instance:

```
new FieldValidator('someField',FieldValidator.verifier.NotBlank);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

When the `onblur` event handler of an instrumented field is invoked, the `validate()` method is called. The definition of this method, which does most of the work of the class, is shown in listing 10.15.

Listing 10.15 The `validate()` workhorse method

```
validate: function() {
  this.clearError();                                  ❶ Clears errors
  if (this.verifier instanceof Function) {
    var message = this.verifier(this.field);          ❷ Verifies JavaScript function
    if (message != null) this.markInError(message);
  }
  else {
    var validator = this;                             ❸ Initiates Ajax request to URL
    var paramHash = {};
    paramHash[this.options.paramName] = $F(this.field);
    new Ajax.Request(
      this.verifier,
      {
        parameters: $H(paramHash).toQueryString(),
        method: 'get',
        onSuccess: function(transport) {
          if (transport.responseText != '') {
            validator.markInError(transport.responseText);
          }
        }
      }
    );
  }
},
```

This method first calls the `clearError()` method that clears the error ❶. This is done in the case that the field has previously failed validation in order to restore it to nonfailure status. More on that in a little while.

It then performs one of two operations depending on the nature of the verifier. If the verifier is a JavaScript function ❷, the function is called, and if it returns anything other than null (indicating a failure), it calls a method that places the field in error status.

If the verifier is a URL, an Ajax request is initiated to that URL ❸, passing the field's value. If the response from that request is anything other than the empty string (indicating a failure), again the field is placed in error status.

The `markInError()` method, shown in listing 10.16, is called whenever a field fails validation either by a client-side or by a server-side check. Because the same

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

action is taken in either case, users are presented with validation failures in a consistent manner regardless of where the check took place.

Listing 10.16 Marking fields as failed

```
markInError: function(message) {
  Element.addClassName(this.field,
                       this.options.errorClassName);
  this.errorMessageElement = document.createElement('div');
  this.errorMessageElement.appendChild(
    document.createTextNode(message));
  this.errorContainer.appendChild(
    this.errorMessageElement);
  Element.show(this.errorContainer);
},
```

This method performs two operations. First, it adds the style class name (recorded in the options) for failed fields to the list of class names for the field. This will cause the field to take on whatever appearance the page author determines appropriate for fields that have failed validation.

Second, it adds a `<div>` element containing the validation failure message to the container identified for this purpose. Since the container is initially hidden, it also ensures that the container is visible.

The final method in our class undoes, or clears, the field from error state, as shown in listing 10.17. This method removes the error style class from the field, and also removes the error message from the message container.

Listing 10.17 Clearing failed fields

```
clearError: function() {
  Element.removeClassName(this.field,
                          this.options.errorClassName);
  if (this.errorMessageElement) {
    this.errorMessageElement.parentNode
      .removeChild(this.errorMessageElement);
  }
}
```

This class is a good example of why we organize code into classes in the first place. Placing this kind of code directly on each and every page that we want to use it on would be nothing short of madness!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

You may have noticed a theme running through the solutions in this chapter, as well as elsewhere in the book. That theme is, "Keep the goo off the pages!" where "goo" is a precise and scientific term for *unnecessary complexity*.

In fact, let's take a look at how little "goo" needs to be put on a page in order to use our field validator.

Using the FieldValidator class

Let's imagine another form in our recipe-sharing community application, this one a simple form presented to visitors who wish to become members of the site. This form is set up similar to the recipe entry form of our previous solution, including instrumentation with contextual help, but possesses only three fields: member name, email address, and age.

The first two fields are required, with the email field restricted to validly formatted email addresses. The age field is optional, but if provided, must be a numeric value.

This example HTML page can be found in the downloadable code for this chapter in the file `/solution-10.4/solution-10.4.html`. When displayed, the page looks like figure 10.8 once you tab out of the name field and enter a bogus

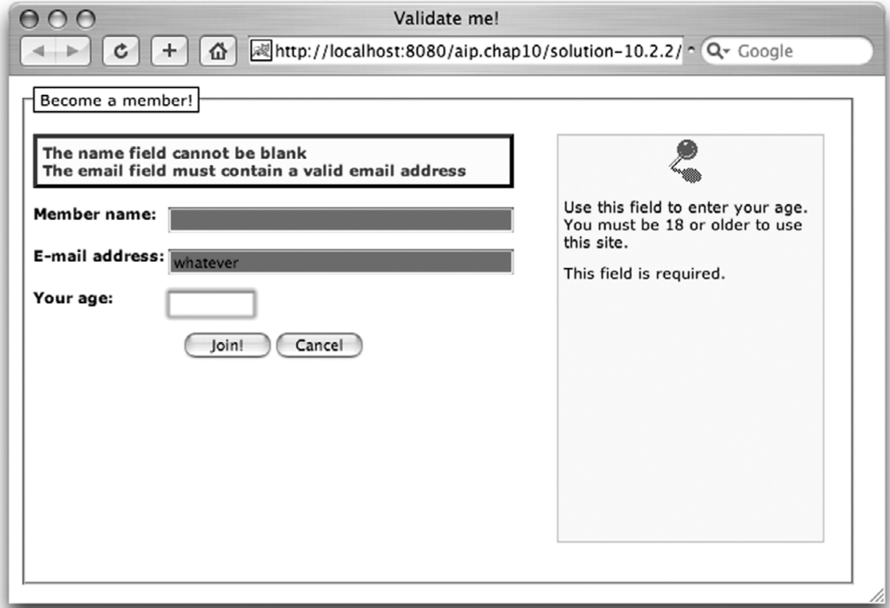

Figure 10.8 Don't forget your name!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

email address. (Note that in the actual code for this solution, the style class for fields in error displays the same yellowish background as the error message box. But as that did not show up well in the grayscale screen grabs, a more strident red was used to grab the screen shot for the figure.)

Wishing to use the FieldValidator class to validate these fields as they are entered, we add the `onload` event handler shown in listing 10.18 to our page.

Listing 10.18 Instrumenting validation for the membership form

```
window.onload = function() {
  new FieldValidator('nameField',
                     FieldValidator.verifier.NotBlank);
  new FieldValidator('emailField',
                     '/aip.chap10/command/VerifyEmail');
  new FieldValidator('ageField',
                     FieldValidator.verifier.IsNumeric);
  new HelpConveyer(
    'helpDisplay',
    document.memberForm,
    '/aip.chap10/command/GetHelp',
    'topic'
  );
  document.memberForm.name.focus();
}
```

In this handler, we set up a validator instance for each field in the form. The name and age fields make use of the built-in client-side verifiers, while the email field uses a server-side resource to check the format of the email address.

That's all there is to it. The remainder of the event handler sets up a new instance of the HelpConveyer class as described in the previous solution, and sets the focus to the name field upon page load.

The only additional task is to add a container in which to display validation failure messages. At the top of the form we define it as

```
<div id="errorContainer" style="display:none;"></div>
```

We give it the following style to make sure it can't be missed:

```
#errorContainer {
  border: 3px outset maroon;
  background-color: #ffffcc;
  padding: 4px;
  color: maroon;
  font-weight: bold;
  margin-bottom: 12px;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

As we mentioned at the beginning of this section, we've laid some good groundwork for a field validator, but much remains to be done. Primarily, we'd need to add the ability to easily find out if there are any fields in error so that we can prevent form submission until all fields validate correctly.

We could also add more built-in verifiers. How about one that verifies a numeric range?

It could also be argued that ganging the messages up into one location (we used the top of the form) would not work well for longer forms. Contemplate how you would change the class to allow validation messages to appear near the field that they are describing.

But perhaps the biggest flaw in what we've set up so far is that this class, as well as the HelpConveyer of the previous solution, usurps the `onblur` and `onfocus` event handlers exclusively, preventing them from being used for any other purpose by page authors. This is a direct result of using the DOM Level 0 Model event handling for simplicity. Using the information that you gathered about the DOM Level 2 Event Model in chapter 5, how would you go about changing these classes to remove the restrictions?

We went through a lot of trouble to make sure that validation failures were reported in a consistent fashion to the user regardless of whether the check occurred on the client or server. That's good, but we can only do so much. If there is a long latency when server-side validation takes place, the user is going to be aware of a difference. It could be more disconcerting than helpful if a field gets placed into an error state after the user has already tabbed several fields beyond it. In cases where network latency makes such progressive validation untenable, it might be best to stick with client-side checks until the form is submitted.

Now that we've helped to eliminate confusion stemming from data entry issues, let's take a look at ways that we can nip other sources of confusion in the bud.

10.3 Maintaining focus and layering order

We already mentioned that there is little that can frustrate a user as much as confusing them. In this section we'll examine ways that we can avoid unnecessary user confusion and frustration in the face of the dynamic user interfaces that we can present to them, now that we have Ajax in our toolbox.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

10.3.1 Maintaining focus order

In a point-and-click world, focus management may not seem like all that interesting a topic. But in the Ajax world, in which user interfaces can be quite dynamic, it may become a critical part of keeping our applications usable.

Let's say that for reasons critical to our application, form fields or other user interface elements need to be shuffled around, hidden and revealed, or even moved about under control of the user. In such cases, the tab order of the input elements (the order in which the elements are visited upon a press of the Tab key) may get rather badly muddled.

And when the tab order becomes nonintuitive, our application becomes considerably less usable. This is especially true for those users who, because of physical restrictions or preference, favor the keyboard over the mouse.

Problem

We want to keep the tab order of user interface elements intuitive, even when those elements are moved about.

Solution

In this section, we're going to use some rather nontraditional user interface elements: `<div>` elements with image backgrounds. Normally, our user interface elements would be text fields and the like, but there's not a lot to distinguish one text field from the other, so for illustrative purposes we'll be using something with a little more texture so that we can readily see the results of our machinations.

Unfortunately, this means that this is the only example in this chapter that will not execute properly in Safari, which currently does not seem to allow `<div>` elements to gain focus.

Our solution page will consist of a series of six elements, each displaying a different image so that we can readily identify them. We'll allow these elements to be rearranged, and see what we need to do in our script to keep the tab order of the rearranged elements intuitive. When first displayed, our page looks as depicted in figure 10.9.

Each item depicts an image, along with a number in the upper-left corner showing its tab order. When the page was loaded, the tab order of the items was assigned 1 through 6, from left to right. The apple initially has focus (we know by the dotted line that the browser draws around the element), and if we were to hit the Tab key successively, we'd see that the mushrooms, then the carrots, then the chiles obtain the focus sequentially, just as we (and more importantly, our users) would expect.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

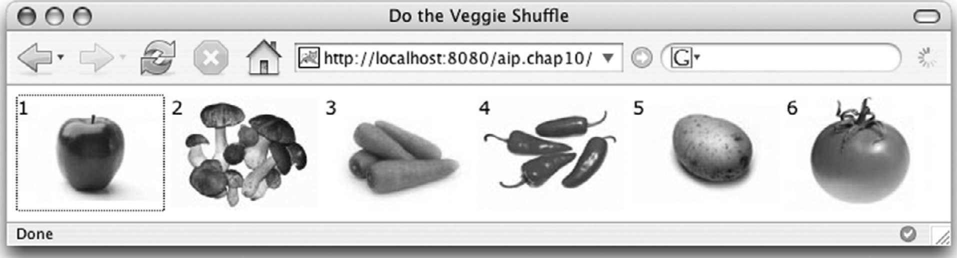

Figure 10.9 Delicious items all in order

We've also instrumented each item so that, when it has focus, the left and right arrow keys can be used to modify its order. The left arrow will cause the item to be swapped with the item to its left, while the right arrow will swap the item with its rightmost neighbor. After playing with this feature for a bit, we might end up with a display like the one in figure 10.10.

Now that we've moved all the items around, notice that the tab order has gone haywire! With focus on the apple, pressing the Tab key brings us back to the mushrooms, then skips over the chiles to the carrots, then back to the chiles, then way over to the potato, and so on. How confusing!

It behooves us to change the tab order of the items whenever we move them so that the tab order remains in a canonical, and therefore, intuitive sequence. What we *really* want after shuffling our produce around is what's shown in figure 10.11.

Now the tab order matches the physical order of the items, and no one should be surprised when they hit the Tab key to proceed to the next item (or Shift+Tab to revert to the previous item).

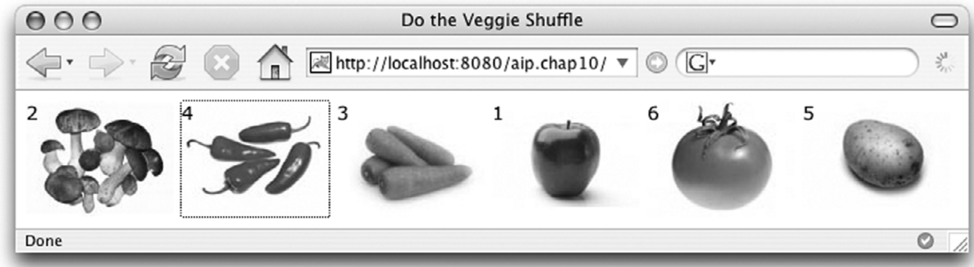

Figure 10.10 Mixed vegetables!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

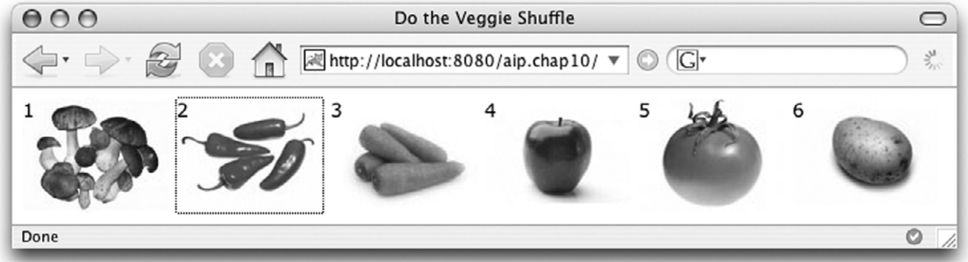

Figure 10.11 Mixed but orderly veggies

So how'd all that take place? And how is the tab order manipulated? Let's take a look at the page's code. It's easiest to start with the item elements themselves, so let's examine the page's `<body>` element as shown in listing 10.19.

Listing 10.19 Creating the produce items

```
<body>
  <div id="container">
    <div id="apple"
         style="background-image:url('apple.jpg');"></div>
    <div id="mushrooms"
         style="background-image:url('mushrooms.jpg');"></div>
    <div id="carrots"
         style="background-image:url('carrots.jpg');"></div>
    <div id="chiles"
         style="background-image:url('chiles.jpg');"></div>
    <div id="potato"
         style="background-image:url('potato.jpg');"></div>
    <div id="tomato"
         style="background-image:url('tomato.jpg');"></div>
  </div>
</body>
```

The `<body>` element contains an outer `<div>` container (used more for styling than for anything else) that contains the six tab-able items. Each item is assigned a unique `id`, as well as its background image. Note that we did not assign the tab order or the little "tab order indicator" that will appear in each item's upper-left corner. Since we're going to be arranging them under script control, we'll set all that up in code. Styles applied via CSS to the items give them absolute positioning, as shown in listing 10.20.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 10.20 Styling the produce

```
<style type="text/css">
  #container {
    position: relative;
  }
  #container div {
    position: absolute;
    width: 110px;
    height: 86px;
  }
</style>
```

Since we're also going to be moving the items around, their actual location is not specified by CSS, but will be set up in script upon page load. Let's take a look at the `onload` event handler for the page, shown in listing 10.21.

Listing 10.21 The onload handler and friends

```
var items = ['apple','mushrooms','carrots','chiles',
             'potato','tomato'];                          <-- (1) Defines array of id values

window.onload = function() {                        (2) Arranges items
  arrange();                                        <--
  items.each(
    function(item,index) {                                <-- (3) Assigns onkeydown handlers
      $(item).onkeydown = move.bind($(item));
    }
  );
  $(items[0]).focus();
}

function arrange() {
  items.each(                                       (4) Arranges each item in order
    function(image,index) {                         <--
      $(image).style.left = (116 * index) + 'px';
      $(image).tabIndex = index + 1;
      $(image).innerHTML = $(image).tabIndex;
    }
  );
}
```

First we set up an array with the `id`s of all the items (1). Note this is not the best of design decisions. We've done it here for expediency and to keep the page centered on focus management; however, having the list repeated in this array and by the items in the `<body>` element is poor design, as they could easily get out of

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

sync with each other. A better design would generate the array from the items, or generate the items from the array.

The `onload` event handler for the page immediately calls the services of the appropriately named `arrange()` function in order to arrange the items ❷. We'll see how it does that in just a moment.

It then iterates over each item, assigning it an `onkeydown` event handler ❸. This handler will be responsible for detecting the arrow keys and for shifting the position of the items when appropriate. Finally, we assign the focus to the first item at the end of the `onload` event handler.

The `arrange()` function, called when the page loads, will also be called whenever the order of the items is shifted. It iterates over the array of item `ids` ❹ and performs three tasks on each:

- It assigns the item its location. Since each item is 110 pixels wide, assigning multiples of 116 gives them 6 pixels of breathing space between them.
- It assigns the tab order in the same order as the `ids` order in the array. Since the tab order should start with 1 but array indexes begin with 0, the value 1 is added to the array index to compute the tab order value.
- The `innerHTML` property of the item is assigned the value of its `tabIndex` property, causing the value of the tab index to appear in the upper-left corner of the item.

Each item was assigned a key handler when the page loaded. That handler function is shown in listing 10.22.

Listing 10.22 Handling the keys

```
function move(event) {
  if (!event) event = window.event;
  if (event.keyCode == 37) {
    moveItem(this,-1)
  }
  else if (event.keyCode == 39) {
    moveItem(this,+1)
  }
}
```

When this handler is invoked as a result of a keypress while an item has focus, the event information is passed to the handler as its parameter—at least in standards-compliant browsers. Internet Explorer 6, loath to stoop to standards, has

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

its own idea of how event handling should occur. To deal with that, we check if the parameter was supplied, and if not, grab the event structure from the window instance where IE 6 insists on putting it.

The key code for the key that was pressed is examined, and if it was the left arrow (key code 37) a function named `moveItem()` is called, passing the function context (`this`) and a value that specifies how the item is to be moved—in this case, back by one.

If the key code identifies the right arrow (39), the item is moved one spot forward.

Note that the function context object for this handler is the item itself. This was because we had the foresight to use the Prototype `bind()` function when assigning the handler. Without this binding, the function context would have been the window.

The work of moving the item is performed in the `moveItem()` function, which is shown in listing 10.23.

Listing 10.23 Moving the item

```
function moveItem(item,by) {
  var oldIndex = item.tabIndex - 1;
  var newIndex = (oldIndex + by + items.length) % items.length;
  items[oldIndex] = items[newIndex];
  items[newIndex] = item;
  arrange();
  item.blur();
  item.focus();
}
```

The function in listing 10.23 moves the passed item by shifting its position *in the `items` array* rather than by physically moving the item itself. Remember, the `arrange()` function takes care of the location and the tab index based on the `items` array, so all we need to do is to rearrange the array as we'd like and let `arrange()` do all the dirty work for us.

So, this function swaps the item with its neighbor according to the value of the `by` parameter. Note the use of the modulus operator to allow items at either end of the list to wrap around to the other end of the list. Once the array has been shuffled, we call `arrange()`, which physically relocates the items and assigns canonical tab indexes to them.

The final task performed in this function looks rather odd. What we've done is rearrange the tab order of the elements behind the browser's back, which

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

leaves it rather confused. It might be argued that the browsers should be able to handle this situation cleanly, but they generally don't. So we basically force a browser to wake up and get its bearings by blurring focus away from the item and forcing it back.

Discussion

While pictures of vegetables (OK, apples and tomatoes are fruits) don't represent realistic user interface elements, this solution used them to ensure that we could clearly see what happens when focusable items are rearranged on the page.

We saw that rearranging items without reassigning their tab order can lead to disconcerting behavior, and we saw the type of scripting code necessary to readjust the tabbing order of the elements.

Although this example didn't use Ajax per se, its lesson is important in Ajax web applications, as dynamic shifting of display elements is often part of a rich user interface design.

10.3.2 Managing stacking order

There is likely to come a point in one of our web applications when we must be able to layer elements on top of other elements. Whether we are dragging and dropping, writing a wizard that has multiple virtual pages, creating floating modal dialog boxes, or writing our own window manager in JavaScript, it all boils down to the same thing: manipulation of the *stacking order*. In HTML pages, the stacking order is defined by the z-index, which defines how "high up" an element is. The higher the value of the z-index, the more "on top" (closer to the user) a layer is. Thus, an element with a z-index of 100 will be drawn on top of an element with a z-index of 99. When elements have the same z-index, they are stacked in the order in which they are syntactically declared in the HTML.

As with focus management, dealing with stacking order is an important lesson for Ajax developers who are creating rich user interfaces, even if it's not directly an Ajax topic.

Problem

We want to learn how to control the stacking order of elements in our rich web application pages.

Solution

We're going to borrow the produce from the veggie drawer again to learn how to manipulate the stacking order of page elements via their z-index. Let's make a

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

page similar to the one of our previous solution, except this time instead of laying the items out linearly, we're going to overlap them. And instead of paying attention to tab index, we're going to focus on the z-index. Our page initially displays as shown in figure 10.12.

Not only have we changed the arrangement of the items to overlap, we've added a border around them (to make their area clearer) and the number in the upper-left corner of each item represents the z-index of the item. As we can see, the higher the z-index, the more "on top" the item is drawn.

We've also added an `onclick` handler to each item that causes it to become the top element by changing its z-index to 6, while readjusting the z-index of the remaining elements.

After clicking the carrot (with an initial z-index of 3), the page appears as shown in figure 10.13.

If we were then to click on the apple, the z-index, and therefore stacking order of the items, would rearrange again, resulting in figure 10.14.

The code to accomplish all of this is similar to the code of the previous solution—so much so, that we're not going to inspect it piece by piece. The entire code for the page is shown in listing 10.24, and we'll just point out a few interesting points regarding changes from the tabbing index example of the previous section.

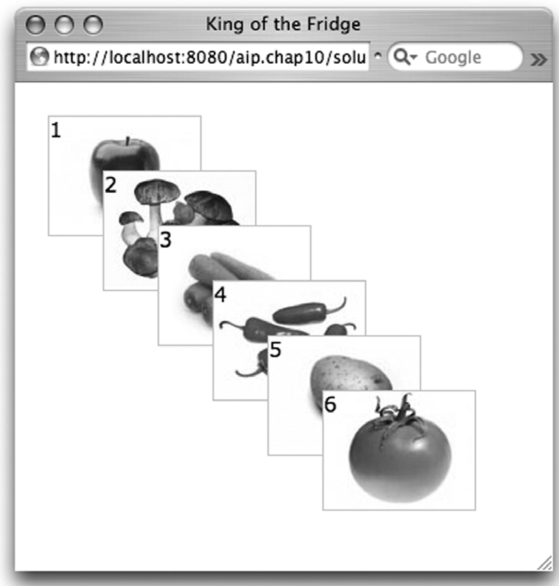

Figure 10.12
The tomato is the king of the crisper.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

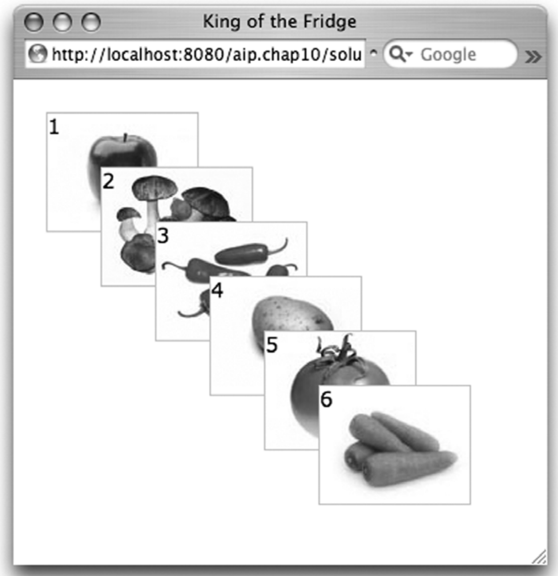

Figure 10.13
The carrot takes control.

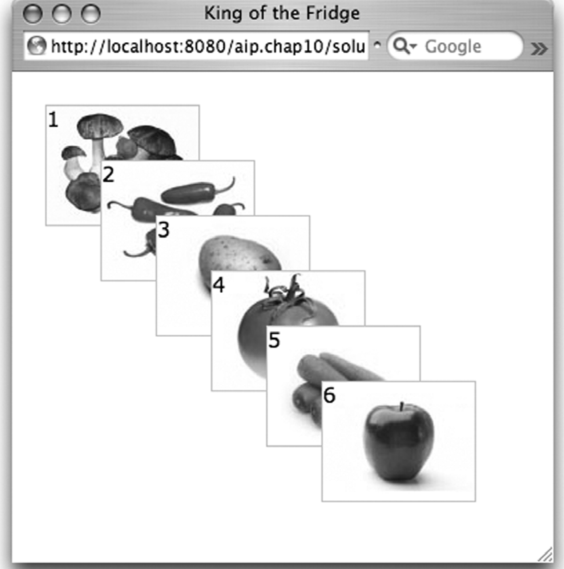

Figure 10.14
The apple asserts its dominance.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 10.24 Controlling the stacking order with z-index

```
<html>
  <head>
    <title>King of the Fridge</title>
    <script type="text/javascript" src="../prototype-1.5.1.js">
      ➥ </script>
    <script type="text/javascript">
      var items = ['apple','mushrooms','carrots','chiles',
                   'potato','tomato'];

      window.onload = function() {
        arrange();
        items.each(
          function(item,index) {
            $(item).onclick = raiseItem.bind($(item));      ❶ Adjusts z-index on mouse click
          }
        );
        $(items[0]).focus();
      };

      function arrange() {
        items.each(
          function(image,index) {                            ❷ Positions items
            $(image).style.left = (16 + (40 * index)) + 'px';
            $(image).style.top = (16 + (40 * index)) + 'px';
            $(image).style.zIndex = index + 1;
            $(image).innerHTML = $(image).style.zIndex;
          }
        );
      }

      function raiseItem() {                                 ❸ Raises item to top
        var itemIndex = this.style.zIndex - 1;
        var newItems = [];
        items.each(
          function(item,index) {
            if (index != itemIndex) newItems.push(item);
          }
        );
        newItems.push(this);
        items = newItems;
        arrange();
      }
    </script>
    <link rel="stylesheet" type="text/css" href="../styles.css"/>
    <style type="text/css">
      #container {
        position: relative;
      }
      #container div {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        position: absolute;
        width: 110px;
        height: 86px;                              ❹ Adds borders
        border: 1px silver solid;            <┘
      }
    </style>
  </head>

  <body>
    <div id="container">
      <div id="apple"
           style="background-image:url('apple.jpg');"></div>
      <div id="mushrooms"
           style="background-image:url('mushrooms.jpg');"></div>
      <div id="carrots"
           style="background-image:url('carrots.jpg');"></div>
      <div id="chiles"
           style="background-image:url('chiles.jpg');"></div>
      <div id="potato"
           style="background-image:url('potato.jpg');"></div>
      <div id="tomato"
           style="background-image:url('tomato.jpg');"></div>
    </div>
  </body>

</html>
```

The `onload` event handler of this example assigns each item an `onclick` event handler (rather than a key handler) so that their z-index will be adjusted on a click of the mouse ❶.

The `arrange()` function is still used to position the items ❷, but instead of laying them out linearly, it overlaps them. It also assigns the z-index of the item (by assignment to the `style.zIndex` property) according to its order in the `items` array. Note that the numeral assigned as the content of the item is its z-index.

The `raiseItem()` function, established as the `onclick` event handler for the items, reorders the items by creating a new array in which the clicked item is placed in the last position after collecting all the other items into the new array ❸. This new array replaces the old, and the `arrange()` function is called to do its thing. The result is that the items are repositioned and drawn with their new stacking order.

The only other change of note is the addition of a style to draw the border around the items ❹.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

This solution borrowed liberally from the solution of section 10.3.1 to demonstrate manipulation of the z-index of items in order to affect their stacking order. There's nothing too complicated to grasp about this concept: higher z-indexes mean higher stacking order. At least that's the theory. There is a common situation in which IE 6 rears its ornery head to defeat our best-laid plans that involves the `<select>` element.

Let's say that we add a `<select>` element to the end of our page positioned so that it overlaps with our items, and that we assign it a z-index of 4. Because it has the same z-index as the chiles but is defined afterward, we'd expect it to be drawn on top of the chiles. We'd also expect it to be drawn under the potato and the tomato, as those items have z-index values higher than 4.

Indeed, that is what occurs in most well-behaved browsers. As shown in the left portion of figure 10.15, the page displays correctly in Safari. IE 6, on the other hand, insists on drawing `<select>` elements on top of everything regardless of any z-index values, as shown on the right portion of the figure. IE 6 will also exhibit this behavior with any `<iframe>` elements.

There are a couple of tactics used to get around this deficiency in IE 6 besides rearranging the design of the page such that `<select>` elements never overlap with anything else. One method often employed is to hide select elements (using

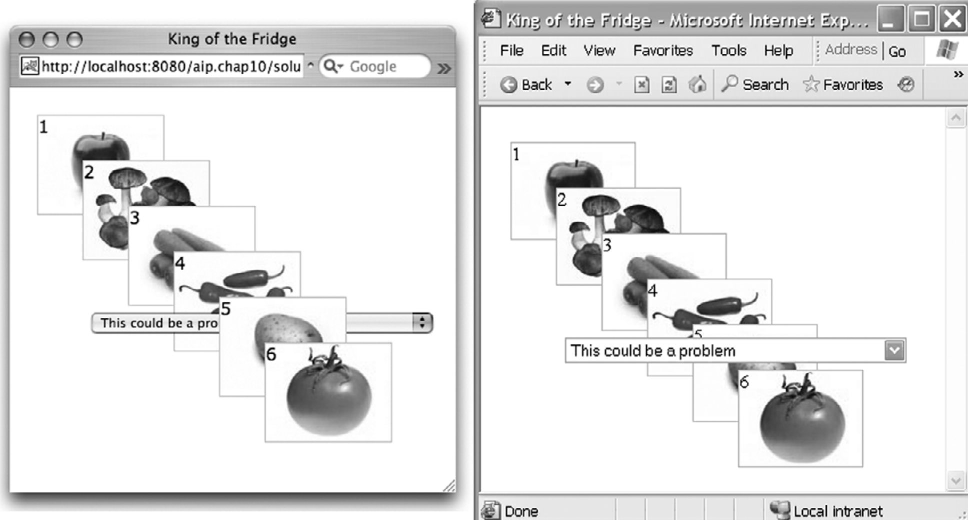

Figure 10.15 Hmm. This could be a problem.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the CSS `display` rule) when something is to be displayed over it. Another tactic is to make the overlapping element an `<iframe>` element in which the higher layered elements are displayed.

Neither of these solutions is very satisfactory, and the latter has a severely invasive effect on the pages as portions of the same logical page need to be segregated out into other HTML documents so that they can be loaded into an `<iframe>`.

Fortunately, Internet Explorer 7 does not exhibit this particular issue.

10.4 Summary

In this chapter we've discussed how the same attention to usability that is customarily applied to desktop applications should also be applied to our web applications—especially as they become more complex and rich with the addition of Ajax to our toolbox.

Usability can be enhanced with rather minor factors that we just need to be mindful of. Keeping the user informed, and presenting interfaces that are intuitive and never surprise the user by doing something unexpected, can win most of the battle.

We've seen some ways that we can achieve those goals using some simple techniques that Ajax makes possible. These techniques were presented using an object-oriented approach that helps to keep the extra code that these techniques add to our pages from creating an unmaintainable mess.

Delighting, rather than frustrating, our users should be a goal of every web application developer. With Ajax in our corner, that objective is more attainable than ever.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

11 State management and caching

This chapter covers

- Maintaining client state
- Prefetching server data
- Saving data locally on the client

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Ajax applications tend to be more client-based than the previous generation of web-based applications. Whereas before state and data would be handled mainly by the server through the use of server-based session data and server-side data processing and display, the advent of the rich client has led to more and more data being managed by the client. Take, for example, Google Mail. This application is extremely dependent on the client for its layout and program flow. The server acts mainly as a repository for information, and the client maintains the data and displays it.

Precisely because the client is now more involved with the handling of data, we need to pay more attention to several things that have changed between the old way of doing things (application mainly on the server; data handling mainly on the server) and the sparkling new Ajax way (much of the application on the client; data handling mainly on the client). Primarily, the way data is loaded is different. Previously, the server would load the data from a database, render it, and send HTML to the client. Now, data can be loaded separately by the client and can be rendered there as well. This can lead to the generation of many small request-and-response cycles instead of just one big request-render-response cycle.

One thing to worry about here is the latency involved with the loading of data from the database. Non-Ajax applications would usually cache the data on the server; subsequent requests for data could then be answered with cached data from the server. Clever server-side data caching strategies could be implemented because the server knew what the client was about to do next in the application, since most of the application was on the server. Now, with the server-side component of Ajax applications more stateless and much of the application residing on the client, it can be tough to implement clever server-side caching strategies because we don't know what the client can do next. Thus, client-side caching becomes increasingly important to reduce the inherent latency in requesting data from a server.

As you might expect, a lot of issues come into play when dealing with client-side data, all of which need to be rethought when dealing with Ajax applications. You, as a web developer, will need to think about

- Security
- Data consistency
- Performance

Because client-side state management is mainly done for performance reasons, we'll mostly concern ourselves with optimizing our applications for speed.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

11.1 Maintaining client state

When dealing with web applications, you may encounter two variations of the problem of maintaining user state on the client side. The first involves maintaining the user's state while they are actively using the application; this problem mainly deals with keeping data in memory across the pages in the application. The second involves maintaining user state across user sessions.

The first problem can be easily solved by maintaining a *data frame* as part of a frameset (listing 11.1). Basically, your application runs in two frames. One frame is visible and takes up the entire browser; it is the frame in which the user interacts with the application. The second frame is invisible and is only used as a repository of data. Why the need for this data frame? Can't you just keep the data in global JavaScript variables? No, you can't, because when you move away from the current page, the variables that you declared on that page disappear. When you navigate back to that page, all your JavaScript variables are once again squeaky clean. Thus we use the data frame to hold our data between navigations in the application. More precisely, global JavaScript variables that point to our data are maintained on the data frame.

Let's look at an example of the behavior we are talking about. We'll quickly list the `<frameset>` declaration (listing 11.1), followed by the frame contents (listings 11.2 and 11.3), before we move on to their discussion.

Listing 11.1 Frameset declaration

```
<html>
<frameset cols="0%, 100%">
  <frame noresize src="data_frame.html" name="data" />
  <frame src="content_frame.html" name="content" />
</frameset>
</html>
```

Listing 11.2 Data frame

```
<html>
<script type="text/javascript" language="javascript">
 var foo = 'foo';

 function getfoo() {
   foo = foo + '1';
   return foo ;
 }
</script>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 11.3 Content frame

```
<html>
<body>

<a href="content_frame2.html">foo</a>

<script type="text/javascript" language="javascript">
  alert(parent.data.getfoo());
</script>
</body>

</html>
```

The `content_frame2.html` referenced in the content frame is identical to `content_frame.html`, except that it points back to `content_frame.html` in the anchor tag. Now, as we navigate through the application (this simply swaps between content frames 1 and 2) by clicking the links, we'll see that we keep getting alert boxes, claiming something about "foo" appended by an increasing number of 1s.

This is obviously a simple example, but it can create some interesting behavior. An example that comes immediately to mind is the simple yet powerful "wizard" concept. In a wizard application, the user needs to navigate through multiple pages of settings. The values of these settings must be retained as you navigate through the wizard. The usual way of doing this is to keep sending the data to the server as the user navigates through the pages, and then retaining this data in memory somehow (in the J2EE world, this would be in the user's session). Then when the user finishes the last wizard page, all the data must be combined into some sort of transaction. When we use the data-frame approach, the values entered in the wizard are kept in the data frame as the user goes through the steps, and are sent to the server only at the final step. This removes some of the complexity of maintaining wizard state from the server and puts it on the client.

The data-frame approach is not useful if your application resides on a single page and uses client-side layout management to handle dynamic content updates. In that case, you can just use global JavaScript variables to hold handles to your data. In addition, a drawback of the data-frame approach is that it will not persist data in JavaScript variables across a refresh of the page. If your users like to click the refresh button a lot, they will keep losing their state. You also need to keep a security issue in mind: if you are using the data-frame approach, be aware that the data stored in these frames is available to all the windows that a browser

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

may have open. It is possible for malicious persons to craft web pages that can trawl through the data retained by the data frames. This data could then be sent back to a server of their choice.

The second problem we mentioned earlier involves maintaining user state when a user logs out of an application and closes the browser. This can be handled on the server side by maintaining all user state data in a database. It can also be handled on the client side by somehow persisting the data on the user's computer. JavaScript applications running on the client side only have one browser-native way to persist data on the browser side: cookies. Certainly, we could resort to ActiveX or Java applets to take care of data persistence on the client side (such plug-ins have access to the client's filesystem), but that would require users to install these plug-ins and goes against the notion of a lightweight client-side application. The reader is naturally encouraged to explore such methods, and we'll examine one such heavyweight client-side persistence mechanism: AMASS (Ajax Massive Storage System). The only reason we showcase AMASS is because it utilizes the ubiquitous Flash plug-in (according to the AMASS authors, available on 95 percent of machines) for storage. In section 11.3 we'll explore state persistence via cookies and AMASS in greater depth.

If we would allow non-cross-browser-compatible ways of storing large amounts of data on the client side, we could take a look at Internet Explorer's client-side persistence mechanism. A lot of documentation is available from Microsoft at http://msdn.microsoft.com/library/default.asp?url=/workshop/author/persistence/overview.asp. Be careful if you do decide to use Microsoft's client-side storage mechanism, as it is quite limited; only about 640KB of custom storage is available per domain and 64KB per page.

11.2 Caching server data

A slow user interface is almost guaranteed to evoke user aggravation. And what is a major cause of a slow user interface? It's the delay incurred when requesting data from the server and getting it back. Thus, to provide a snappy client-side interface, we must be able to obtain the data to display quickly. There are several ways to do this on the server (such as tuning your queries or caching commonly requested data), but a request-response overhead is still incurred. The only way to get rid of this overhead is to keep your data on the client. This section will explore this aspect of Ajax applications: server data caching.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We'll take a look at how we should store server data on the client side. Naturally, we must still obtain the data to store, and so we'll also examine how to use the prefetching of server data to speed things up even more.

You might be wondering how you can cache the data you receive from the server. The answer to this question is highly dependent on the type of application that you have. Some possibilities for storing data are

- As JavaScript objects
- As multiple arrays of data
- As XML DOM trees, which can be swapped in and out of the client-side DOM
- Insert your favorite mechanism here

If your data is highly static, it might make sense to prerender them into DOM trees on the client, and then swap them in and out using the DOM. Multiple arrays make sense if you'll be making several modifications to the data and iterating over them. JavaScript objects can be assembled into a complicated client-side caching framework, which can intercept events that happen on said objects and dispatch these events to certain registered listeners (which can then take appropriate actions based on the nature of the events). As you may have deduced, it all depends on the nature of your application.

In this section, we'll store our data in the form of JavaScript objects and use object-oriented JavaScript. Object-oriented JavaScript is becoming prevalent in the Ajax world (and is made easy through the use of the excellent Prototype library), and it thus behooves the developer to become familiar with it.

The first example will build a simple object storage mechanism that reflects the rows retrieved from database queries. The second example will then expand on the first by including a prefetching mechanism. To showcase the functionality of these concepts, we'll develop a simple customer display application where we can page through a list of customers. Let's get coding!

11.2.1 Exchanging Java class data

In this problem we'll examine a simple client-side object storage mechanism to store rows retrieved from a database. We'll start off with a small web page that can do a few simple things. It will be able to query the server for lists of customer information, which it will then store on the client side. It will also be able to page through this data. It won't prefetch anything, nor will it check the server for outdated data. We'll examine those scenarios later on.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

You need to cache server-side data on the client to speed up the interface.

Solution

Let's develop a class for the customers first (see listing 11.4). The end result should look like figure 11.1. The class will hold some simple attributes such as a first and last name, along with a customer ID number. This is simply the index of the customer in an array of customers that we are using to mock up a database. If you were using a real database, this might be a universally unique identifier (UUID) or some other identifying field. For exchanging the data in the Java classes, we'll use the JSON libraries to serialize them to JavaScript.

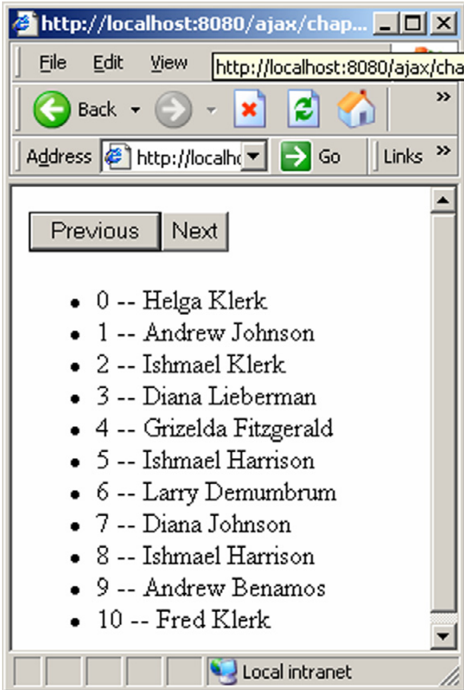

Figure 11.1 Caching customer data on the client

Listing 11.4 Customer class

```
public class Customer
{

    private String firstName;
    private String lastName;
    private int customerID;

    public Customer(String firstName, String lastName, int customerID) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.customerID = customerID;
    }

    public JSONObject toJSON() throws JSONException {
        JSONObject obj = new JSONObject();
        obj.put("firstName", firstName);
        obj.put("lastName", lastName);
        obj.put("customerID", customerID);
```

❶ Converts Java object to JSON

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            return obj;
        }
    }
```

As you can see, we use the JSON Java library (available at www.json.org/) to convert a Java object to its JSON representation ❶.

Now that we have a representation of a customer, we need a class that will store our customers and is able to query them (see listing 11.5). We also need to create some mock data, which this class will also do for us.

Listing 11.5 CustomerManager class

```
public class CustomerManager
{

  private static CustomerManager instance = new CustomerManager();

  public static CustomerManager getInstance() {
    return instance;
  }

  private CustomerManager() {
    this.customers = new ArrayList<Customer>();      <- Creates customer database
    createCustomers(100);      <- Populates database with random customers
  }

  private List<Customer> customers;      <- Stores customer database
  private void createCustomers (int total) {
    Random random = new Random();

    String[] firstNames = { "Andrew", "Benjamin", "Chris",
        "Diana", "Elaine", "Fred", "Grizelda", "Helga",
        "Ishmael", "Julia", "Kevin", "Larry", "Mallory" };
    String[] lastNames = { "Andersen", "Benamos", "Costa",
        "Demumbrum", "Evans", "Fitzgerald", "Glen",
        "Harrison", "Ibrahim", "Johnson", "Klerk",
        "Lieberman", "Murakami" };

    for (int id = 0; id < total; id++) {
      String firstName = firstNames[random
          .nextInt(firstNames.length)];
      String lastName = lastNames[random
          .nextInt(lastNames.length)];
      Customer customer = new Customer(firstName, lastName,
          id);
      this.customers.add(id, customer);
    }
  }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    public List<Customer> getCustomers() {      <-- Queries database for customers
      return this.customers;
    }

  }
```

Now that we have a customer representation and some sort of customer database, we need a way to return the customers stored in the database to our JavaScript running on the client. The CustomerServlet shown in listing 11.6 will take care of this.

Listing 11.6 CustomerServlet class

```
public class CustomerServlet extends HttpServlet
{

  CustomerManager cm = CustomerManager.getInstance();     <-- Obtains customer manager

  @Override
  protected void doGet(HttpServletRequest req,
      HttpServletResponse res) throws ServletException,
      IOException {
    int start = Integer.parseInt(req.getParameter("start"));   <-- Specifies customer start index
    int pageSize = Integer.parseInt(req                 <-- Specifies total customers to return
        .getParameter("pageSize"));

    List<Customer> customers = cm.getCustomers();

    JSONArray ja = new JSONArray();                  <-- (1) Creates new JSON array
    for (int current = start; current < start + pageSize
        && !(current >= customers.size()); current++) {
      try {
        ja.put(customers.get(current).toJSON());      <-- (2) Stores customer
      } catch (JSONException e) {
        e.printStackTrace();
        throw new ServletException(e);
      }
    }

    res.getWriter ().write(ja.toString());       <-- Writes representation to client
  }
}
```

Note that we use a `JSONArray` (1) to hold a JavaScript representation of all the customers we'll return. We then put a JSON representation of the customer into the array (2) we'll return to the client.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Our application is beginning to take form as all our server-side components are ready. All that is left is to create some JavaScript code that can query the server for customers and display them on a page. We'll write a CustomerManager class (see listing 11.7) in object-oriented JavaScript with Prototype to handle the fetching, caching, and displaying of the customers.

Listing 11.7 Client-side CustomerManager

```
var CustomerManager = Class.create();

CustomerManager.prototype =
{
  customerData : new Array(),                    ❶ Holds customer data

  drawCustomerDIV : function(start, pageSize, div, cached) {
    displayString = '<ul>';

    for (i = start;
         i < pageSize + start && i < this.customerData.length;
         i++) {
      customer = this.customerData[i];
      displayString += '<li>';
      displayString += customer.customerID;

      if (cached) {
        displayString += ' (cached) ';          ❷ Displays cached notification
      }
      displayString += ' -- ';
      displayString += customer.firstName;
      displayString += ' ';
      displayString += customer.lastName;

      displayString += '</li>';
    }

    displayString += '</ul>';

    div.innerHTML = displayString;               ❸ Shows customer data
  },

  cacheCustomerData : function (response) {
    responseArray = response.responseText;
    currentCustomerData =
      eval('(' + responseArray + ')');           ❹ Deserializes fetched
                                                   customer data
    for (i = 0; i < currentCustomerData.length; i++) {
      customerID =
        currentCustomerData[i].customerID;       ❺ Gets customer ID
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        this.customerData[customerID] =
          currentCustomerData[i];     ◁─ ❻ Caches customer data
      }
    },

    getCustomerData : function (start, pageSize, div) {
      if (this.customerData.length > start) {
        this.drawCustomerDIV(start,              ❼ Draws customer data
          ➥ pageSize, div, true);             ◁─
      } else {
        manager = this;     ◁─ ❽ Manages reference
        options = {              to self for callback
            method: 'get',
            parameters: 'start=' + start +      ❾ Passes proper
              '&pageSize=' + pageSize,       ◁─    parameters to server
            onSuccess: function(response) {
              manager.cacheCustomerData(response);
              manager.drawCustomerDIV(start, pageSize, div, false);
              },
            onFailure: function(r) {
                alert('Server Status: ' + response.status + ' - ' +
                response.statusText);
            }
        };

        new Ajax.Request('/ajax/servlet/Customers', options);

      }
    },

    initialize : function() {}
  }
```

Let's take a closer look at our CustomerManager class. First, we create an array to cache the customer data ❶. This array will be used extensively in the subsequent code.

Second, we have our `drawCustomerDIV()` function, which takes the cached customer data and creates a list in the `<div>` that we specify. The `displayString` variable will hold the HTML of the list of customers we're assembling. Once we've assembled the contents of the `<div>`, we can set the content of the `<div>` to the HTML code contained in the `displayString` variable. The `cached` parameter will be `true` if we did not fetch the customers we're displaying from the server; instead we'll rely purely on the cached data in the `customerData` array. In this case, we append a notice that these customers are from the customer cache ❷.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Finally, we set the content of the `<div>` ❸ we're using to display the customers to the list that we have assembled in the loop.

Third, we have our `cacheCustomerData()` function, which is invoked only after we've fetched customers from the server. It will create JavaScript objects out of the JSON array we've received from the server and will populate the `customerData` array appropriately. It gets the JSON text we've received from the server and creates a JavaScript array out of it ❹. Then, for each customer object in the array, we get its ID ❺ and store it in the cache indexed by its ID ❻.

Finally, we create the `getCustomerData()` function, which is capable of deciding whether the requested range of customers is in the customer cache. If it is ❼, the function will simply output the customer data. If the range is not held in the cache, the function will request the customer objects from the server, cache them, and then draw them. We also store a reference to the manager ❽ because later in our `onSuccess` event handler we reference the `manager` variable instead of the `this` variable. The reason is that when this callback function is invoked, the `this` variable won't point to the CustomerManager object and will instead point at an object in the Ajax.Request object hierarchy. Thus, we need to invoke the manager like this. Because we're using Prototype's Ajax.Request object to create an asynchronous XHR object, we populate the options object to pass several options to Ajax.Request. In the options, we pass a `start` parameter ❾, which denotes the index in the database to start at, and a `pageSize` parameter, which denotes the number of customers to return. If the request is successful, we'll cache the customers from the response and draw them. The last action we perform is to make a request to our data servlet. The path `/ajax/servlet/Customers` is where our CustomerServlet is located.

Displaying the customer data

We're almost done, as we have the server- and client-side code. Figure 11.2 shows what it looks like when we receive notification that the currently displayed customers are being fetched from the internal customer cache instead of being fetched from the server.

Next up is some HTML (listing 11.8) for displaying the customer data along with two JavaScript functions that will allow us to page through the data.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

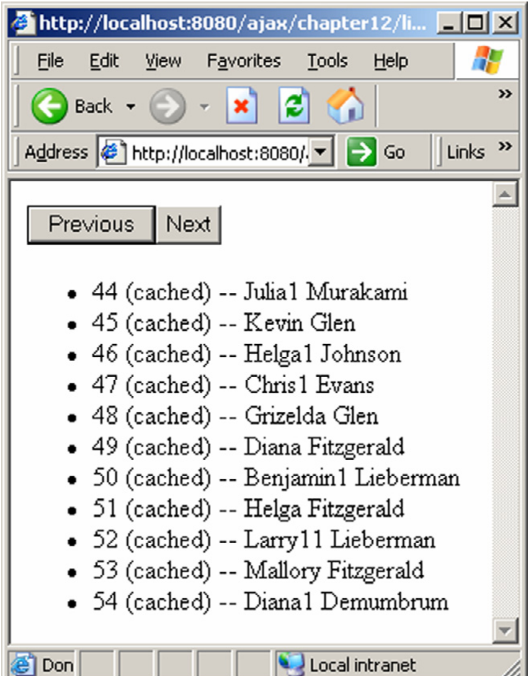

Figure 11.2 Displaying the cached customers

Listing 11.8 Customer paging HTML

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
   <script type="text/javascript" src="../prototype-1.4.0.js"></script>
   <script type="text/javascript" src="listings.js"></script>
   </head>
<body>

  <button onclick="previous();">Previous</button>
  <button onclick="next();">Next</button>

  <div id="customers"></div>          <-- Contains <div> for displaying customer data

  <script type="text/javascript">
  var totalCustomers = 100;
  var currentCustomerIndex = 0;
  var pageSize = 11;

  var manager = new CustomerManager();
  function previous() {                <-- Moves to previous customer list
    currentCustomerIndex -= pageSize ;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    if (currentCustomerIndex < 0) {
      currentCustomerIndex = 0;
    }
    manager.getCustomerData(currentCustomerIndex, pageSize,      <-- Displays customers in <div>
    $('customers'));

   }

  function next() {        <-- Moves to next customer list
    currentCustomerIndex += pageSize;
    if(currentCustomerIndex >= totalCustomers) {
      currentCustomerIndex = totalCustomers - 1;
    }
   manager.getCustomerData(currentCustomerIndex, pageSize, $('customers'));
  }

  manager.getCustomerData(currentCustomerIndex, pageSize, $('customers'));
  </script>
</body>
</html>
```

A quick note on this code: we need to know how many customers in total are returned by our query—in this case, *all* the customers. We're hard-coding that value in our `totalCustomers` variable to cut down on the lines of code you need to read through. If you were doing this in the real world, you'd assign this variable programmatically based on the total number of customers that are available for display.

Discussion

We now have all the components that we need. Let's skip ahead a few pages with the Next button, and then hit the Previous button to go back to the previous list. Because we have cached those customers already, we should be informed that those customers are being fetched from the cache.

This section showed a simple example of complex client-side behavior. In the real world, things are not this simple. As we mentioned, we hard-coded the number of customers available for display to cut down on client- and server-side code. In the real world, this variable needs to be fetched programmatically when we execute a query.

Caching previously requested results is a big performance win for both users and application providers. Users enjoy the increased performance boost of not having to round-trip to the server for information they had already loaded. Application providers are able to save on server cycles and bandwidth, as they do not incur extra hits for data they had already sent to the client.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We can optimize the experience for the user even more. In the next example, we'll extend our caching behavior to also load the next page that a user can navigate to. This way, the cache will already be warm, and page flips take absolutely no time at all.

11.2.2 Prefetching

To provide for an even more responsive user interface, we need to examine the topic of prefetching, or precaching. The basic idea is that we know what data the user is going to look at next, as they have only the option initially of going to the next page. Instead of waiting for the user to hit the Next button in order to load that data, we load it and cache it before the user does anything. Due to the asynchronous behavior, this prefetching takes place in the background and will not cause the client to experience a lag as the browser is grabbing the future data.

Problem

You need to prefetch data from the server to speed up the client.

Solution

We'll be reusing much of the previous example with a few modifications made in the appropriate areas. The server-side code doesn't need be changed one bit. It simply serves our customers; this behavior is no different now than it was before. The only place where we need to make modifications is in our client-side Client-Manager object. Instead of making one request to the server, it will now make either two (when we first load up the page and to get the current page and the next page) or one (when we hit Next, it needs to get the second-next page). The result should look like figure 11.3. Let's look at the modified ClientManager JavaScript (listing 11.9).

Listing 11.9 Client-side JavaScript

```
drawCustomerDIV : function(start, pageSize, div, cached) {
    if (div == null) {          <-- (1) Prevents infinite recursion
      return;
    }

    displayString = '<ul>';

    for (i = start;
         i < pageSize + start && i < this.customerData.length;
         i++) {
      customer = this.customerData[i];
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        displayString += '<li>';

        displayString += customer.customerID;
        if (cached) {
          displayString += ' (cached) ';
        }
        displayString += ' -- ';
        displayString += customer.firstName;
        displayString += ' ';
        displayString += customer.lastName;

        displayString += '</li>';
      }

      displayString += '</ul>';
      div.innerHTML = displayString;
    }


  getCustomerData : function (start, pageSize, div) {
      if (this.customerData.length > start) {
        this.drawCustomerDIV(start, pageSize, div, true);
        if(div != null) {
          this.getCustomerData(start + pageSize, pageSize, null);
        }
      } else {
        manager = this;
        options = {
            method: 'get',
            parameters: 'start=' + start + '&pageSize=' + pageSize,
            onSuccess: function(response) {
              manager.cacheCustomerData(response);
              manager.drawCustomerDIV(start, pageSize, div, false);
              if (div != null) {
                manager.getCustomerData(
                   start + pageSize, pageSize, null);
              }
            },
            onFailure: function(r) {
                alert('Server Status: ' + response.status + ' - ' +
   response.statusText);
            }
        };

        new Ajax.Request('/ajax/servlet/Customers', options);

      }
    }
```

2 Detects call by UI

3 Detects call by UI

4 Gets next page of data

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

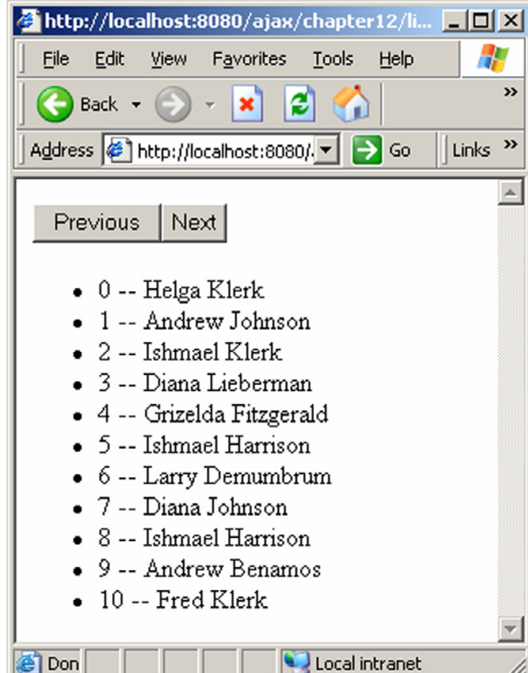

Figure 11.3
Initial screen with non-prefetched customers

Discussion

We only make one modification to the `drawCustomerDIV()` function ❶: if the `<div>` is `null`, do not do anything. You will see in the next function why that is so.

We make a few modifications to the `getCustomerData()` function. If the `<div>` is not `null` ❷ (meaning we were called directly from the UI), then we get the next page of customer data and pass `null` as the `<div>`. This is so because we do not wish to draw the next page, and because we want to prevent infinite recursion. Likewise, if the `<div>` is not `null` ❸, we fetch the next page of data ❹. We pass `null` as the value for the `<div>`, because we need to prevent recursion and do not wish to draw the prefetched page.

And that is all. A few small modifications and we are now prefetching the data. We expect the first page to not display the *cached* notification (figure 11.3), but any subsequent pages we navigate to should display the notification.

Now let's go to the next page (figure 11.4), which should tell us that it used the cache for the customer data. Indeed we see that these customers were fetched from the cache and not from the server.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

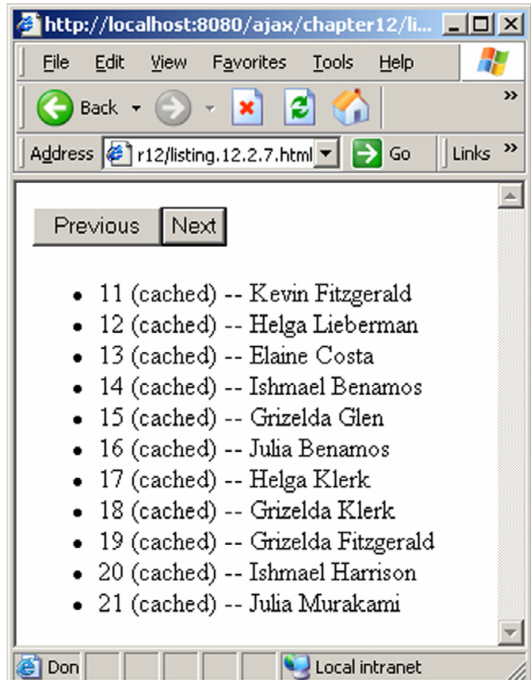

Figure 11.4
Browser displaying the cached notification for prefetched customers

It was quite simple to modify the CustomerManager to prefetch the data: we made only three small changes. In return, we are now prefetching the next page of customer data in the background while the user is inspecting the current data.

An interesting problem presents itself: what if the user is paging through the application faster than the rate at which we can precache? With the previous solution, the proper data will still be displayed: as we navigate to the next page, we will detect that we do not have the customers in the cache (because they are still being fetched), and thus another request to the server is issued for those customers. This can lead to a situation where we are making multiple requests for the same data—most certainly a suboptimal situation. If we were using Java (or any other programming language with mutex abilities), we need simply implement a locked reader-writer, where the reader will wait on a mutex if the data is not available and be awoken by the writer once the information is present. Sadly, JavaScript does not have any such synchronization capabilities, which means we're out of luck. There are certainly ways around the problem; one possible solution is to implement a poor man's threading approximation:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

- The *page forward* and *page backward* capabilities are implemented as command objects, which are placed on a command queue. They can check if the needed data is present. If it is present, we update the UI; if it is not present, then we fail.
- We use the `setInterval()` or `setTimeout()` method to periodically execute a command-object executer. The command-object executer would look at the command object queue and execute the topmost object. If it succeeds, we remove it from the execution stack. If it fails, we leave it alone until the next time we enter our execution loop.

The problem could also be solved on the server side. The server could detect identical queries and place any duplicate queries in a wait queue. Once the original query returns, it could pass the same results to the identical queries. However, this way you are still exchanging duplicate data between the server and the client.

As you can see, it isn't an easy problem to solve. Is the solution worth the additional complexity? That depends on the application.

11.3 Persisting client state

The previous section dealt with transient data—that is, data that is not persisted on the client across browser restarts and page refreshes. This section explores how we can permanently store data locally on the client machine.

As we discussed earlier, there are two ways to store data locally on the client: via cookies and through browser plug-ins that can access the local filesystem. We'll examine both of these mechanisms in the examples that follow.

11.3.1 Storing and retrieving user state with JSON

This example will show you how to store/retrieve a JSON representation of the user's state to/from cookies. We are storing the user state in a tree of JavaScript objects and will be using the JSON JavaScript library to serialize and deserialize the data. You can find more information about JSON at www.json.org. The JavaScript JSON library can be found at www.json.org/js.html. For storing and reading cookies, we are using the Webmonkey cookie library available from www.webmonkey.com/webmonkey/reference/javascript_code_library/wm_ckie_lib/. There are about 12.5 billion cookie libraries out there, so don't feel obligated to use this one.

Problem

You need to store and retrieve data in a cookie across browser restarts.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Solution

Let's start this solution off with a mix of JavaScript and HTML (listing 11.10), which shows you how to combine JSON data and cookies. Nothing too complicated here; it's just a simple script that stores the information from a small IP address wizard in a browser cookie.

Listing 11.10 Serializing and storing a state object using JSON and cookies

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
<script type="text/javascript" src="../cookies.js"></script>
</head>

<script type="text/javascript">
function getState() {                          <-- Gets representation of user state
  mainState = new Object();
  var wizardState = new Object();

  mainState.wizard = wizardState;

  wizardState.ip = '192.168.1.6';
  wizardState.nm = '255.255.255.0';
  wizardState.gw = '192.168.1.1';

  mainState.userName = 'json';
  mainState.password = 'nosj15';
  mainState.style = 'bluesteel';

  return mainState;
}

function storeState(state) {                   <-- Stores state
  serialized = state.toJSONString();           <-- Converts to a JSON string
  WM_setCookie('state', serialized, 24*100);   <-- Sets expiration to 100 hours
}


function testStateMechanism()
{
  state = getState();                          <-- Gets current state
  storeState(state);                           <-- Stores state in cookie
}

testStateMechanism();                          <-- Tests our mechanism
</script>
</html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

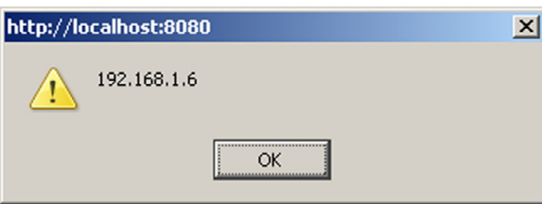

Figure 11.5
Alert box showing our retrieved state

When we run listing 11.10, we store the current state in a cookie. Then we close the browser, start it back up, and run listing 11.11. Figure 11.5 shows us what we get for our trouble.

Listing 11.11 Deserializing a state object using JSON and cookies

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
<script type="text/javascript" src="../cookies.js"></script>
</head>

<script type="text/javascript">
function retrieveState() {
  serialized = WM_readCookie('state');      <- Reads serialized data from cookie
  return serialized.parseJSON();            <- Returns deserialized data
}

function testStateMechanism()
{
  state2 = retrieveState();
  alert(state2.wizard.ip);
}

testStateMechanism();
</script>
</html>
```

Discussion

There are a few limitations (as specified in RFC 2109) to using cookies to store data across browser restarts. First, cookies are limited to 4KB of data. That's not a whole lot if you wish to store, say, the contents of your customer relationship management database. Second, you are limited to 20 cookies per domain, so even if you were to persist state across multiple cookies, you're limited to 80KB of data.

Also, keep in mind that all the data stored in cookies for a particular domain will be sent to the server. You must be sure that once you read the user state from

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

a cookie, you remove that cookie from the client; otherwise, each client request will send across the contents of the state management cookie. This will certainly slow down your applications.

Finally, a security consideration is that users can easily modify the cookie contents. You must be careful not to store sensitive data in a cookie, such as usernames, passwords, logged-in status, roles, groups, and so forth. The security consideration is twofold:

- Users of the same machine could access this data to steal passwords and other sensitive information and impersonate the valid user by simply copying the cookie.
- If information such as roles and groups are stored in the cookie, users could manipulate this data to grant themselves rights that they should not have.

Therefore, all matters pertaining to login and security should never be stored on the client side.

11.3.2 Persisting JSON strings through AMASS

Cookies simply will not do the trick when you wish to store more than 80KB of information on the client. What is a web developer to do? AMASS (http://codinginparadise.org/projects/storage/) springs into action and comes to the rescue. It makes use of the Flash plug-in to store large amounts of data on the client, which is saved to local files. You can store a total of 100KB without user permission. Any larger amounts require the user to grant permission to Flash to store this amount. AMASS works a bit like a hash table: data is stored by a specific key. You can save simple text strings, and even JavaScript objects, which are simply converted into a string representation.

Problem

You need to store and retrieve large amounts of data on the client.

Solution

Let's reimplement the previous example using AMASS (listing 11.12).

Listing 11.12 Using AMASS

```
<html>
<head>
<script type="text/javascript" src="../json.js"></script>
<script type="text/javascript" src="x_core.js"></script>
<script type="text/javascript" src="x_dom.js"></script>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
<script type="text/javascript" src="x_event.js"></script>
<script type="text/javascript" src="storage.js"></script>
<script type="text/javascript" language="javascript">
storage.onload(initialize);                    <-- Registers event handler

function initialize(){
  alert('store is initialized');
}

function storeState(state) {                   <-- Encodes object to JSON and stores it

  storage.putString('state', state.toJSONString(), statusHandler);
}

function testStateMechanism() {                <-- Retrieves state and prints info
  state2 = retrieveState();
  alert(state2.wizard.gw);
}

function retrieveState() {                     <-- Retrieves stored state, converts to object
  return storage.getString('state').parseJSON();
}

function persistState() {                      <-- Gets application state and persists it
  state = getState();
  storeState(state);
}

function getState() {                          <-- Creates mock state with nested objects
  mainState = new Object();
  wizardState = new Object();

  mainState.wizard = wizardState;

  wizardState.ip = '192.168.1.6';
  wizardState.nm = '255.255.255.0';
  wizardState.gw = '192.168.1.1';

  mainState.userName = 'json';
  mainState.password = 'nosj15';
  mainState.style = 'bluesteel';

  return mainState;
}

function statusHandler(status) {
  if (status != Storage.SUCCESS) {             <-- Alerts if not successful
    alert(status);
  }
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    </script>
    </head>
    <body>
     <button onclick="persistState();">
        Store State</button>
     <button onclick="testStateMechanism();">
        Test Stored State</button>
    </body>
</html>
```

Discussion

As we discussed previously, any in-memory state is lost when we navigate away from a page or when we refresh the browser. Thus, to test AMASS and show its functionality, we'll load the example page, store the state, refresh the page, and test the state. Let's do that.

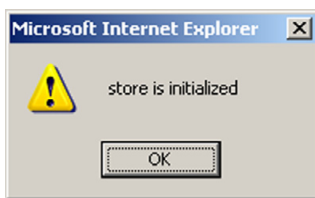

Figure 11.6 AMASS initialization message

First we are greeted with an alert box (figure 11.6) stating that the storage system has been initialized. This is generated from the callback we passed to `storage.onLoad()`. We also see the main application page with our store and restore buttons (figure 11.7).

Let's click the Store State button and refresh the page. This should generate another alert box notifying us that the storage system is ready. After dismissing the alert box, we then click the Test Stored State button. This should pop up another alert box (figure 11.8) notifying us of an element of the persisted state.

And indeed it does. The value displayed in the alert box (192.168.1.1) is the value of the `gw` variable we assigned to the wizard state, which was then assigned to the global state holder. In our test function, we navigated the restored state to the wizard object, and then to the `gw` value, and then displayed it in our alert box. As you can see, by combining AMASS with JSON we can store entire object trees and load them again when needed.

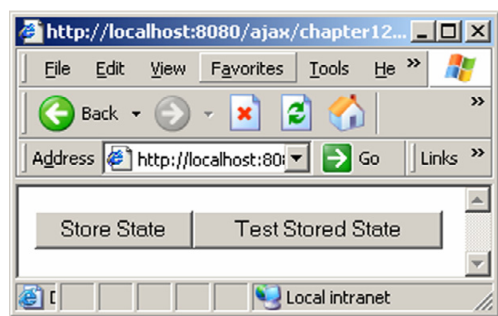

Figure 11.7 Storage test

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

You may wonder where the `storage` variable that we used in the previous JavaScript comes from. It is actually declared by the storage.js JavaScript library. Once you have included the AMASS JavaScript library on your page, you can then just refer to it anywhere in your JavaScript functions. AMASS also has other functions that you may find useful. We've been using the `putString()` function to store our objects, but AMASS also has a `put()` method that can take any object, not just strings. However, in our testing we've noticed that it is faster to do the manual JSON encoding first and then use `putString()` than it is to just pass a nonencoded JavaScript object to the `put()` method. Likewise, `getString()` has an equivalent `get()` counterpart, which will deserialize objects stored with the `put()` method. But just as JSON + `putString()` is faster than regular `put()`, JSON + `getString()` is faster than a regular `get()`. AMASS also has a function called `hasKey()`, which simply returns `true` or `false` depending on whether a specific key has an associated value. Conspicuously absent is a `delete()` function! When we needed to remove a key, we simply called `put()` with `null` as argument; this seems to work just as well.

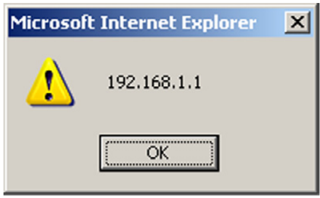

Figure 11.8 Storage test success

Like cookies, AMASS stores its data in files on the filesystem. And also like cookies, these files are quite easily manipulated by a person with a text or hex editor and some spare time. The security considerations we've mentioned for cookies also apply to using AMASS for your persistent storage. Do not store sensitive information using AMASS.

AMASS is an extremely capable tool when you have a need to persist large amounts of information on the client side. For instance, a webmail application could store read messages on the client, obviating the need to refetch emails once they have been downloaded. A drawback of using client-side persistence is that upgrades of your software are not as simple as they previously were. If an application is purely on the server, you are able to manipulate data any way you choose. Now that a significant amount of data is stored on the client, you lose that advantage and you are faced with writing a client-side program that can upgrade the data stored on the client. Caveat programmer!

Dirty-checking cached data

One matter that we did not discuss in the previous examples is the *dirty-checking* of data held by the client against the data from the server. You will run into situations where data is updated on the server and your client is holding a cached

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

copy of the old data. What to do? You could implement complicated caching algorithms that ask the server whether any of its data is outdated and that instruct the server to send the client any updates. To provide this functionality, the server and client need to keep some sort of timestamp or version number for the data being cached by the client. This is so that we can compare the timestamp or version number with the same information held on the client. Then, the client must communicate to the server the timestamp information it has and receive updates to any modified data.

However, when you check client-side data against the server in order to ensure that the data is consistent, a large assumption is made: it is faster to check and fetch only updated data than it is to just fetch the server-side data no matter what. The validity of this assumption depends on your application. If it takes a long time to obtain the latest data, then this method will work for you. If it is just as fast to access the latest data as it is to check the data for updates and only retrieve those updates, then you may be better off not implementing a complicated version-checking mechanism: it will simplify your application without causing a performance hit. In most instances it will be faster to just fetch the new data: you are querying the data for their latest timestamps anyway, so you might as well just avoid the hassle of writing complicated client- and server-side code. You most likely will not achieve performance gains anyway. Thus, if your application must display the latest and greatest data at all times, you are probably better off not caching anything on the client. On the other hand, if it takes a long time to fetch your data and timeliness is not quite important, it is best to cache on the client and perhaps refresh the cache periodically; with XHR, this could occur in the background without the user being aware of it.

11.4 Summary

In this chapter, you learned how to store transient data (typically database data that is updated frequently) and persistent data (for example, user settings and application state). We showed you two methods for each: with transient data, you can cache server data or use a similar method that involves prefetching; with persistent data, you can persist state to cookies (for small amounts of data) or use the AMASS library (for large amounts of data).

You can—and should—use data caching in your own applications. Using data caching will make your applications much faster as a result of decreased network round-trip and server access time. Using data caching also reduces the load on your servers (both application and database) by a substantial amount.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Beware of the security problems with client-side caching. If your users work with sensitive data, like credit card or social security numbers, you may want to take a good look at how you will employ caching techniques. Your data might be compromised if your users' computers or laptops are stolen—leaving you with some explaining to do. The Open Web Application Security Project (www.owasp.org/index.php/Guide_Table_of_Contents) has quite a lot of things to say on the topic of web application security, and you may find the information there quite useful.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

12 Open web APIs and Ajax

This chapter covers

- Using open web APIs
- Creating a cross-server proxy
- Using the Yahoo! Maps, Geocoding, and Traffic APIs
- Using the Google Search API
- Using Flickr photos APIs

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Sometimes you just can't do it all.

As web developers, we're used to writing most of the code that needs to be created in order to deliver the specific applications and projects that we take on. But if you stop and think about it, in addition to the code that we write ourselves, we rely on a vast array of support software that does a lot of work for us. From the web servers that deliver our pages, to the browsers and all their supporting software, to the operating systems that the servers and client browsers themselves run on, all this serves as a supporting framework that we depend on when writing our own applications.

Beyond such enabling technology, there is a dizzying array of frameworks and libraries that help us add features to our applications. One of the most exciting of these is the area of *open APIs*, (application programming interfaces) where some well-known (and a lot of not-so-well-known) websites expose APIs that allow you to integrate the technology that they provide with your own applications.

Imagine an application where you might wish to incorporate maps, or one in which you'd like to search the Web, or one for maintaining and sharing photo collections. Each of these features would be a task of Herculean proportions to implement on our own. But, to our delight, someone's already done it, and they're willing to open their functionality to us.

This is a great advantage to us, the application developers. We can leverage the hard work that others have done by accepting the generosity that they show in opening their APIs for us to use. Granted, some may look at that generosity with a jaded view as a marketing tactic to get developers interested in more capable professional services that may not be free, but as long as the providers are up front and not deceitful, it's a fair marketing practice.

In this chapter we'll look at code examples that integrate Yahoo! Maps, Geocoding, and Traffic; Google Web Search; and Flickr photo services into our Ajax-enabled web applications. In doing so, we'll explore a good cross section of techniques for integrating open APIs into web pages using Ajax.

So let's dig in and have some fun!

12.1 The Yahoo! Developer Network

The Yahoo! Developer Network (http://developer.yahoo.com/) provides a large range of web services, including travel, shopping, jobs, and much, much more. To use many of these services, you are required to obtain an application key from Yahoo!. This application key is a string of your choosing (much like a username), and it is usually passed as a request parameter to any call that you make to the Yahoo! web service APIs to identify who is making the call.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Links in the various documents available at the Yahoo! Developer Network guide you through obtaining your application key. The URL for the application key request page is http://api.search.yahoo.com/webservices/register_application/. Note that you must be logged into your Yahoo! account in order to access this page.

The Yahoo! servers will ensure the uniqueness of each key during registration. To be sure that the keys registered are unique, and to make them easy to remember, you can use a scheme similar to the Java package-naming conventions when choosing application keys. Let's say that you own your own domain: `yourlastname.org`. All Java packages you would create in your projects would begin with `org.yourlastname.`*`projectname`*, where *projectname* is the name chosen for an individual project. Similarly, if Yahoo! web services are to be used in that project, you would register an application key named `org.yourlastname.`*`projectname`*. This will usually guarantee uniqueness unless someone is horning in on your package-naming territory!

Once you've registered a key, you're ready to begin using the Yahoo! web services immediately.

12.1.1 Yahoo! Maps

Let's say that you and a group of your friends are gutsy storm chasers. Every spring during tornado season, you load up your gear and head out in hopes of videotaping exciting footage that you hope may appear on The Weather Channel.

Problem

Your friends have high-end global positioning systems (GPS) that display real-time maps. Alas, your unit is an old, but trusty, instrument that only gives you latitude and longitude coordinates. You don't have the extra cash to plunk down for a more modern unit, but you do have a laptop that connects to the Internet via your mobile phone. You also have programming skills, and you have Ajax!

So let your friends have their fancy GPS units—we'll build our own map application!

Solution

The Yahoo! Maps (http://developer.yahoo.com/maps/index.html) service provides a number of API choices, but we'll stick with the Yahoo! Maps Ajax API to leverage our knowledge of JavaScript and DHTML.

Unlike some other Yahoo! services (which we'll see other examples of in just a bit), the Yahoo! Maps Ajax API wraps the Ajax magic into a ready-built JavaScript API. So rather than making the Ajax calls ourselves, we'll import the Yahoo! Maps API and reference the objects and functions that it defines.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

First, we must import the API from the Yahoo! servers. In the `<head>` section of our document we'll place the following script import element:

```
<script
  type="text/javascript"
  src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
</script>
```

We also import Prototype with

```
<script
  type="text/javascript" src="prototype-1.5.1.js ">
</script>
```

We'll obtain the latitude and longitude coordinates from our GPS unit and enter them in a form on our application page. Upon entering the data in the form (which we will *not* be submitting to any server), we want to display a map centered on those coordinates.

So let's set up a little form as follows (making it beautiful is up to you):

```
<form name="mapForm" onsubmit="showMap();return false;">
  <div>
    Latitude: <input type="text" id="latitude"/>
    Longitude: <input type="text" id="longitude"/>
    <input type="submit"/>
  <div>
</form>
```

Note that the `onsubmit` event handler for this form causes a JavaScript function to be invoked, and also prevents the form from actually being submitted by returning `false`.

In addition to this form, we need someplace for the map itself to be displayed. The Yahoo! Maps API will expect us to pass it an element within which it will "draw" the map, so in the appropriate place on the page let's add an initially empty container:

```
<div id="theMap" style="width:600px;height:480px;"></div>
```

After the document has loaded, we want to create and initialize the map object that is central to the API. We do so in the `onload` event handler of the page since the map container element must exist prior to doing so:

```
var map;
window.onload = function() {
  map = new YMap($('theMap'));
  map.addPanControl();
  map.addZoomLong();
};
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

This sets up a reusable YMap instance that will draw its map into the passed element. We've also added optional controls to the map: a panning control and a zoom control.

When our form is submitted (thus invoking the `showMap()` function), we instruct this YMap instance to draw a map centered at the given coordinates by creating a YGeoPoint instance and passing it to the YMap's `drawZoomAndCenter()` method. This `showMap()` function looks like this:

```
function showMap() {  var zoomLevel = 4;
  var latitude = $F('latitude');
  var longitude = $F('longitude');
  var point = new YGeoPoint(latitude,longitude);
  map.drawZoomAndCenter(point,zoomLevel);
}
```

We chose an arbitrary zoom level of 4, which seems to be a good general starting point. Once the map loads, we can affect the zoom level, since we included a zoom control on the map. But if we wanted to, we could also add another form field to set the initial zoom level to a value other than 4.

If we were to read latitude and longitude values of 30.27 and -97.74 from our GPS unit and enter them into our application, we'd see a display like the one in figure 12.1. Our completed document appears in listing 12.1, and can be found in the downloadable source code for this chapter at www.manning.com/crane2.

Listing 12.1 Yahoo! Maps Page

```
<html>
  <head>
    <title>Where Am I?</title>
    <script
      type="text/javascript"
      src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="prototype-1.5.1.js"></script>
    <script type="text/javascript">

      var map;

      window.onload = function() {
        map = new YMap($('theMap'));
        map.addPanControl();
        map.addZoomLong();
      }

      function showMap() {
        var zoomLevel = 4;
        var latitude = $F('latitude');
        var longitude = $F('longitude');
        var point = new YGeoPoint(latitude,longitude);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          map.drawZoomAndCenter(point,zoomLevel);
        }
      </script>
    </head>

    <body>

      <div>
        <form name="mapForm" onsubmit="showMap();return false;">
          Latitude: <input type="text" id="latitude"/>
          Longitude: <input type="text" id="longitude"/>
          <input type="submit"/>
        </form>
      </div>

      <div id="theMap" style="width:600px;height:480px"></div>

    </body>

  </html>
```

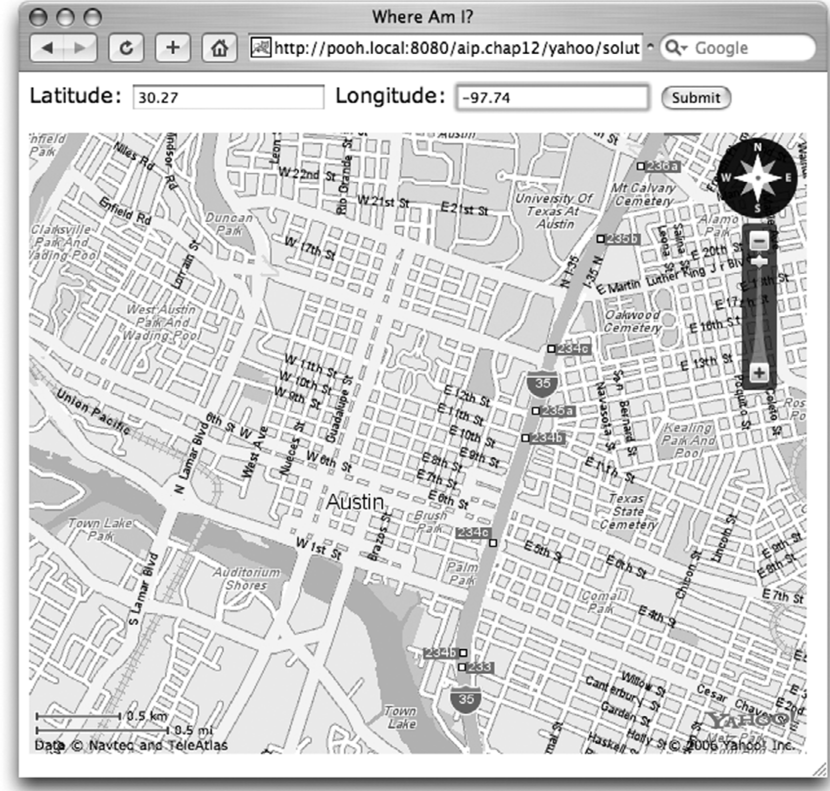

Figure 12.1 Now we know where we are!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

Is that not just too cool? Who needs a fancy GPS?

By just looking at the code on this page, it is not at all clear that there's any Ajax magic going on as the server communication mechanism is hidden behind the Yahoo! Maps JavaScript API. But it's obvious that *some* server somewhere is being contacted to fetch the map data for display.

Also note that this example did not require the use of our newly obtained Yahoo! Developer network application key. But we'll soon see a service that will.

Our little page is perfect for displaying maps given the latitude and longitude coordinates, but sometimes we might have just a plain old street address to contend with. The Yahoo! Maps YMap API has no means of drawing a map given a location address, but we're not going to let that stop us. Yahoo! also makes a Geocoding API available.

12.1.2 The cross-server proxy

The Yahoo! Maps Geocoding REST API (http://developer.yahoo.com/maps/rest/V1/geocode.html) provides the means to determine latitude and longitude coordinates given an address location. Unlike the main Yahoo! Maps API discussed in the previous section, the Geocoding API is a Representational State Transfer (REST) interface.

REST is a simple HTTP interface in which we hit an endpoint URL with appropriate request parameters and receive a response consisting of XML, JSON, HTML, or even plain text without the need of an additional layer such as SOAP (Simple Object Access Protocol).

Fantastic! What could be easier? So we go ahead and code up an Ajax request on our web page that hits the appropriate URL with the required request parameters to test it out, anticipating that we'll receive a response with all sorts of wonderful information.

Instead, the browser reaches out and slaps our wrists saying, "Uh, uh, uh! No cross-server scripting allowed!"

What happened?

Problem

What happened is that we ran head-on into the *Ajax security sandbox*. For security purposes, browsers only allow Ajax calls to the same server that sent the page to the browser in the first place.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Well, dang! That certainly puts a crimp in our plans. If cross-browser scripting isn't allowed, that pretty much slams the door on the use of any REST API via Ajax.

Or does it? Remember, we're clever.

We know that we can make all the Ajax requests that we want to our own server. And once on the server, and outside the realm of the dreaded security sandbox, we can make requests to any other server that we please.

So what we really need is an agent on our own server to act as a proxy on our behalf—making requests to the remote servers that are providing the REST APIs—and to relay the results of those requests back to us.

Solution

We'll explore a Java-based solution that is appropriate for any web application running on servlet containers. The same approach can also be implemented for other server-side technologies, be it PHP or good old CGI scripts in Perl, cURL, or even simple shell scripts employing `wget`!

The idea is to create a proxy agent to which we can make requests and that will relay those requests to another server, collect the response from that remote server, and finally feed the responses back to us. This will be a generally useful utility but it will be particularly valuable to us for circumventing the limitations of the Ajax security sandbox.

Let's take a look at a few diagrams that can help us envision just what this is all about. Consider figure 12.2.

In step 1, our local server, booboo, receives a request (A) from the client, and responds with our HTML page, which the browser loads for display. Upon some event on the page, in step 2 an Ajax request (B) is made to a remote server named yogi that is providing some useful web service that we want to make use of.

But, no! The browser knows that our page was served from booboo, and any attempt to make a request to another server is blocked. Well, we all learned at a young age that when Mom says "no," you go ask Dad! Now consider figure 12.3.

In this new scenario, a request and response cycle (A) to load our page into the browser takes place in step 1 just as before. But now, when we wish to make an Ajax request to the web service on yogi, we make the request (B) back to booboo, the originating server, as shown in step 2. This request is directed at our proxy agent servlet, and contains information that identifies the server and service that we want to contact.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

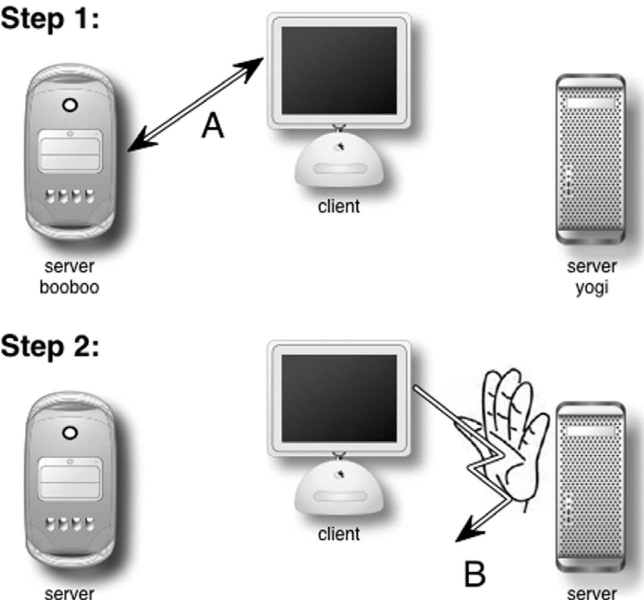

Figure 12.2
Foiled by the Ajax sandbox!

In step 3, the proxy agent servlet makes a request (C) to the remote yogi server, requesting the designated web service. Since we are out of Mom's hearing—in other words, outside the domain of Ajax sandbox—there is nothing to stand in our way, and the remote server returns a response (C) to our request.

Finally, in step 4, the proxy agent server on booboo returns the response received from yogi as its own (B).

Mom is none the wiser!

So let's see what it will take to code up our devious little circumvention trick. We could just jumble all the necessary code into a servlet, but upon a moment of reflection we realize that, since this facility would be generally useful in many environments—command-line programs or daemons, for example, or even in Swing applications—that we should create a reusable component.

So to start off, we'll create a UI-agnostic class that can be used in any Java program to obtain content from a remote server via HTTP.

The content grabber

At this point we are faced with the frequent "build, buy, or borrow" decision. Do we use the facilities of the Java networking packages to "roll our own"? Or has someone else already done a good portion of the work for us?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

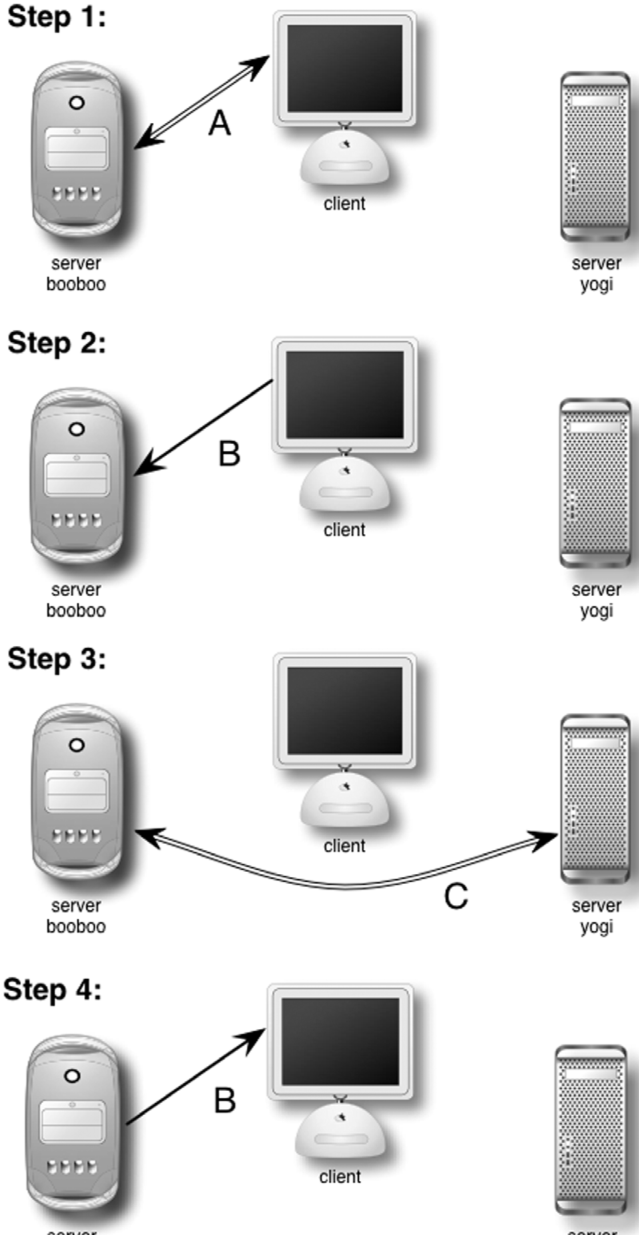

Figure 12.3
We'll just sneak around the back way.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In this implementation of a *content grabber*, we are going to decide to leverage the work already done in this area by other programmers. After all, isn't this entire chapter about leveraging code generously provided by others?

We'll make use of a component of the Apache Jakarta Project named HttpClient. This is an open source component that makes it easy to emulate the actions of an HTTP client—hence its name.

The project information and download can be found at http://jakarta.apache.org/commons/httpclient/. The JAR files necessary to use this component are already available as part of the downloadable source code for this chapter.

Our ContentGrabber class turns out to be fairly simple due to the use of this tool. Its implementation is shown in listing 12.2.

Listing 12.2 The ContentGrabber class

```
package org.bibeault.rest;

import java.util.*;
import org.apache.commons.httpclient.*;                 ❶ Imports HttpClient
import org.apache.commons.httpclient.methods.*;            classes

public class ContentGrabber {

  private String url;
  private String content;                     ❷ Defines instance
  private String contentType;                    variables
  private Integer contentLength;
                                                  ❸ Defines constructor
  public ContentGrabber(String url,
                        Map<String,String[]> parameters) {
    this.url = url;
    try {                                                  ❹ Creates GET
      HttpMethod method = new GetMethod( url );
      List<NameValuePair> params = new ArrayList<NameValuePair>();
      for (Map.Entry<String,String[]> entry : parameters.entrySet()) {
        for (String value : entry.getValue()) {
          params.add(new NameValuePair(entry.getKey(), value));
        }
      }
      method.setQueryString(
        params.toArray(new NameValuePair[params.size()]));     ❺ Executes GET
      new HttpClient().executeMethod(method);
      this.content =
        method.getResponseBodyAsString();
      Header contentTypeHeader =                          ❻ Obtains response
        method.getResponseHeader("content-type");
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        Header contentLengthHeader =
          method.getResponseHeader("content-length");
        if (contentTypeHeader != null)
          this.contentType = contentTypeHeader.getValue();
        if (contentLengthHeader != null)
          this.contentLength =
            Integer.parseInt(contentLengthHeader.getValue());
        method.releaseConnection();
      }
      catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException("Error obtaining content from " +
            this.url + ": " + e, e);
      }
    }

    public String getUrl() { return this.url; }        ❼ Defines accessors
    public String getContent() { return this.content; }
    public String getContentType() { return this.contentType; }
    public Integer getContentLength() { return this.contentLength; }

    public static void main(String[] args) {
      Map<String, String[]> params = new HashMap<String, String[]>();
      params.put("appid", new String[] {"org.bibeault.aip"});
      params.put("location", new String[] {"78701"});
      System.out.println(                    Tests ContentGrabber
        new ContentGrabber                     from command line ❽
          ➥ ("http://api.local.yahoo.com/MapsService/V1/geocode",
                           params)
          .getContent());
    }

  }
```

Our class starts off like any other Java class by importing the external classes that we'll need ❶. We import the HttpClient classes as well as a handful of collections from the java.util package.

A number of instance variables are defined ❷ to hold the input URL, as well as the results of executing the HTTP GET method: the body content, the content type, and the content length. Since not every GET response will include the headers for the content type and length, we need to be sure to handle that. Both of the `contentType` and `contentLength` instance variables are initialized to `null` and will remain `null` if their corresponding header is not returned. Note that we defined the `contentLength` variable as an Integer, rather than an `int`, just so that we can detect whether or not it has a `null` value.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The GET request will be fired off within the constructor ❸ given the passed URL and the optional set of query parameters. Note that we have used Java 5 generics to specify that the passed parameter map be composed of String instances as keys and String arrays as values. In older Java Development Kits, the generics could be removed (but resulting in the loss of the type safety that they provide).

It could certainly be a matter for debate as to whether performing the request as part of construction is an optimum design. It is quite possible that there could be cases where a delay between construction of the grabber and access to its response properties could be desirable. This class could be refactored to perform a "lazy load" whenever one of the property accessors is called. But that won't be an issue for our needs, so we'll just keep things simple for now.

The primary operation of the class takes place by creating an instance of GetMethod to represent the HTTP method that we wish to execute ❹. The created instance of GetMethod is constructed with the passed URL and then augmented with the query parameters after they have been converted from the `parameters` Map to the required array of NameValuePair instances.

When the method is ready for execution, a new instance of HttpClient is created and used to execute the method ❺. The response is inspected ❻ and its body content is recorded in the `content` instance variable. If the content type and length headers were returned, their values are stored in the corresponding instance variables, `contentType` and `contentLength`.

The class defines a number of property accessors to allow callers of the class to obtain the response results ❼.

Finally, the class contains a `main()` method ❽ that can be used to perform a rudimentary test of the class's function. This main method is not used by callers of the class.

Try it out! Load the class into an IDE, or compile and run it from the command line. In either case you should see a printout of an XML document returned by the service.

Now that we have a means to easily grab remote content, let's put it to work.

The cross-server proxy servlet

With the ContentGrabber class at our disposal, we're ready to create the actual proxy agent for our Ajax requests. Since we've already coded all the heavy lifting in the content grabber class, the proxy servlet is actually rather simple, as shown in listing 12.3.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 12.3 The CrossServerProxy class

```
package org.bibeault.rest;

import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class CrossServerProxy extends HttpServlet {

  public static final String KEY_SERVICE_URL = ".serviceUrl.";

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
    String serviceUrl =
      request.getParameter(KEY_SERVICE_URL);
      if (serviceUrl == null) {
       throw new ServletException
                       ("the " + KEY_SERVICE_URL +
                        " parameter must be provided");
    }
    Map parameters = new HashMap();
    parameters.putAll(request.getParameterMap());
    parameters.remove(KEY_SERVICE_URL);
    ContentGrabber grabber =
      new ContentGrabber(serviceUrl, parameters);
    if (grabber.getContentType() != null)
       response.setContentType(grabber.getContextType());
    if (grabber.getContentLength() != null)
      response.setContentLength(grabber.getContentLength());
    response.getWriter ()
      .print(grabber.getContent());
  }

}
```

❶ Supports GET requests
❷ Obtains URL service parameter
❸ Complains if required service URL missing
❹ Removes service URL from copied map
❺ Constructs instance of content grabber
❻ Sets content type and length
❼ Relays content of proxied response

This servlet is shown supporting GET requests ❶. POST support could be easily added via a `doPost()` method that simply calls the `doGet()` method to which it would pass its `request` and `response` parameters.

The servlet accepts a single required parameter ❷ that is used to provide the URL of the request to be proxied. Note that we have chosen a rather unconventional name for the parameter in that it begins and ends with a period character. Why did we do that?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

We don't know in advance what the request parameters for the proxied request will be, so we'll just add any that we find on the incoming request to the proxied request—all, that is, except for the single parameter that we are using to provide the base URL.

That is exactly why we named it with such an odd format. It would be very unusual, and highly unlikely, for a web service that we might be interested in using to define request parameters that contain the period character. Doing so would make HTML DOM elements difficult to reference in JavaScript code as we would be unable to use the dot notation and would need to fall back to using the generalized de-referencing notation employing the square brackets. So by naming the reserved parameter using the periods, we make it unlikely that we would pollute the parameter namespace of any web service that we wish to employ.

We then check that the required service URL parameter has been provided ❸, and complain loudly if it was omitted.

A copy of the parameters passed to the proxy servlet for passing to the content grabber is made ❹. We first copy the request parameter map and then remove the service URL parameter.

Why make a copy? Why not just use the Map instance returned from the `getParameterMap()` method? Remember that Map is an interface, and in that interface, the `remove()` method is an optional method. We have no idea whether or not the implementation of Map returned from the `getParameterMap()` method will implement `remove()`. And in fact, under Tomcat 5.5, the returned Map does not implement it.

By copying the returned Map to a known implementation that implements `remove()`, in this case HashMap, we can guarantee that our call to the `remove()` method will be successful.

An instance of our content grabber is then constructed ❺, specifying the proxy URL and its parameters. This triggers the cross-server call to the proxied URL. Once it completes, the content type and length of the response to the proxied request are recorded in our own response ❻.

Finally, the content of the proxied response is relayed back as the content of our own response ❼.

Discussion

With this solution we have addressed, at least for Java web applications, the problem where the security sandbox prevents us from making web service calls from our pages using Ajax. As mentioned, a similar approach can be used for other server-side environments.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

As with all other code examples in this chapter, the amount of error checking performed in the code is minimal to nonexistent in order to keep the examples focused on the subject at hand. In actual production code, further error checking should be added to increase the robustness of the code.

We created a *content grabber* class that allows us to grab the content from another site given a URL and a set of request parameters. This class relies heavily on the toolset provided by Jakarta's HttpClient project. For those of you who'd rather roll up your sleeves and do it yourselves, check out the java.net.URL and java.net.URLConnection classes as a starting point.

Be aware that this can become a complicated affair. What if, perhaps, a web service that you wish to employ requires cookie handling? That's a complicated affair to add to any handwritten code. HttpClient already contains the tool set necessary to extend our content grabber to expand its functionality in this and similar areas.

The cross-server request *proxy servlet* employs our content grabber to serve as an agent for our on-page Ajax requests. Adding it to a web application is quite simple. Obviously it needs to be placed in the class path of the application. Also, it needs to be declared and mapped in the deployment descriptor (`web.xml`) with two simple elements. The first declares the servlet itself:

```
<servlet>
  <servlet-name>CrossServerProxy</servlet-name>
  <servlet-class>org.bibeault.rest.CrossServerProxy</servlet-class>
  <load-on-startup>4</load-on-startup>
</servlet>
```

The second declares the URL mapping for the servlet:

```
<servlet-mapping>
  <servlet-name>CrossServerProxy</servlet-name>
  <url-pattern>/proxy</url-pattern>
</servlet-mapping>
```

In this case, we have decided to use the servlet path: `/proxy`.

Our next section puts our proxy agent to good use.

12.1.3 Yahoo! Maps Geocoding

Now that we have a means to invoke web services on remote servers thanks to the proxy agent we created in the previous section, we're ready to enhance our mapping solution with the ability to show us a map given either a set of latitude and longitude coordinates or a location address.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Problem

As pointed out previously, the Yahoo! Maps API requires latitude and longitude coordinates in order to draw a map, so if we want to allow location addresses to act as input, we need to convert the location address into its coordinate equivalent.

Solution

The Yahoo! Maps Geocoding REST API http://developer.yahoo.com/maps/rest/V1/geocode.html is the means by which we'll accomplish this. Unlike the Yahoo! Maps API that we employed earlier to display maps given coordinates, the Geocoding API is a REST interface. We'll make a request to the geocoding service URL with appropriate request parameters, and the response will be an XML document with the requested information. The URL for this service is http://api.local.yahoo.com/MapsService/V1/geocode.

Note that at the time of this writing, it appears that this service is limited to U.S. addresses. Please reference the above-mentioned web page for current information regarding supported locations.

We'll send as parameters the required application key that we registered with Yahoo! as well as the location string that we wish to convert to coordinates. The service accepts other parameters, but these are the ones that we'll focus on.

An example service request might be

```
http://api.local.yahoo.com/MapsService/V1/geocode?
  appid=your.yahoo.app.id&location=78701
```

Type that into your browser's address bar and see what you get. Be sure to provide your correct Yahoo! application key. Note that you can provide as little information as a zip code and still obtain meaningful results.

Armed with that knowledge, let's begin the modifications to our map page. First, we need to add a new entry field for the location string. We're going to make this a separate form because we want it to act independently of the existing coordinate form. So before the existing `mapForm` we add

```
<div>
  <form name="geocodeForm"
        onsubmit="findLocation();return false;">
    Location: <input type="text" id="locationField"
                     name="location" style="width:200px;"/>
    <input type="submit"/>
  </form>
</div>
```

This adds a longer text field to our page into which we can type the location string.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Again, since we are going to employ an Ajax call, we prevent the form from actually submitting to the server by returning `false` in the `onsubmit` event handler. The call to the function `findLocation()` is what will trigger the geocoding request:

```
function findLocation() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',                          <- (1) Contains options to Ajax request
      onSuccess: onCoordsObtained,
      parameters: {                           <- (2) Specifies the request parameters
        '.serviceUrl.':
         'http://api.local.yahoo.com/MapsService/V1/geocode',
        appid: 'org.bibeault.aip',
        location: $F('locationField')
      }
    }
  );
}
```

In this function we fire off an Ajax request to the Geocoding API, making heavy use of the facilities that Prototype provides for us. For a function that consists of a single statement, there's certainly a lot going on! Let's take a look at the various aspects.

The URL that we pass to the Ajax request is rather simple: `/aip.chap12/proxy`. Remember that we can't make a direct request to the Yahoo! API, so we route it through our proxy servlet. This URL assumes that the web app has been mapped to the context path `/aip.chap12` and the servlet path `/proxy` routes the request to the proxy servlet.

In the options parameter to the Ajax request (1), we specify an HTTP method of GET and tell it to invoke a callback handler function named `onCoordsObtained()` upon success. We then provide the query parameters to be passed to the request.

Because we are using Prototype 1.5.1 in this example, we can pass the parameters as a simple object hash (2). In earlier versions of Prototype, this hash would need to be converted to a query string. (Rather than laboriously creating the query string by hand, investigate leveraging the facilities of the Prototype Hash class if you are using an earlier version of Prototype.)

Let's take a look at the request parameters that we specified. The `.serviceUrl.` parameter tells the proxy servlet which service to make the request to, the `appid` parameter specifies the Yahoo! Developer Network application key (in this case, one that we registered for use by these examples), and the `location` parameter passes the value that was typed into the location field. Note that our oddly

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

named service URL parameter needs to be quoted because of the period characters within the name.

If all goes well when this request is made, we specified that the callback handler function `onCoordsObtained()` be invoked. It is added as

```
function onCoordsObtained(request) {
  var xml = request.responseXML;
  document.mapForm.latitude.value =
    xml.getElementsByTagName('Latitude').item(0).firstChild.data;
  document.mapForm.longitude.value =
    xml.getElementsByTagName('Longitude').item(0).firstChild.data;
  showMap();
}
```

In this success handler for our Ajax request, we are passed the instance of XHR that Prototype created on our behalf. As a successful geocoding request returns a small XML document as its response, we obtain and peruse that document in order to gather the results of the service request.

A typical XML document returned from a geocoding request that specified the location as simply "78701" would be

```
<ResultSet xsi:schemaLocation="urn:yahoo:maps
http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd">
  <Result precision="zip">
    <Latitude>30.271</Latitude>
    <Longitude>-97.741</Longitude>
    <Address/>
    <City>AUSTIN</City>
    <State>TX</State>
    <Zip>78701</Zip>
    <Country>US</Country>
  </Result>
</ResultSet>
```

For the purpose of this example, the only elements that we care about in this document are `<Latitude>` and `<Longitude>`. Our function locates the data within these elements using the XML DOM API and loads them into the latitude and longitude fields in the `mapForm`.

Finally, our handler calls the existing `showMap()` function to cause the map for those coordinates to be displayed.

Discussion

By making use of the Yahoo! Maps Geocoding REST API, we have added the ability to enter location addresses in addition to coordinates. As a bonus, we display the returned coordinate values in the appropriate forms fields as well as display the proper map.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The code could be made more robust. For example, if the service cannot determine the coordinates for the entered location, it returns a nonsuccess response, and as written, our page just silently ignores it. While it at least does not blow up in our faces, it could probably be smarter about informing us when errors occur. Figure 12.4 shows the revised page.

The complete code for our extended page, which can be found in the downloadable source code for this chapter, is shown in listing 12.4 (with changes and additions in bold).

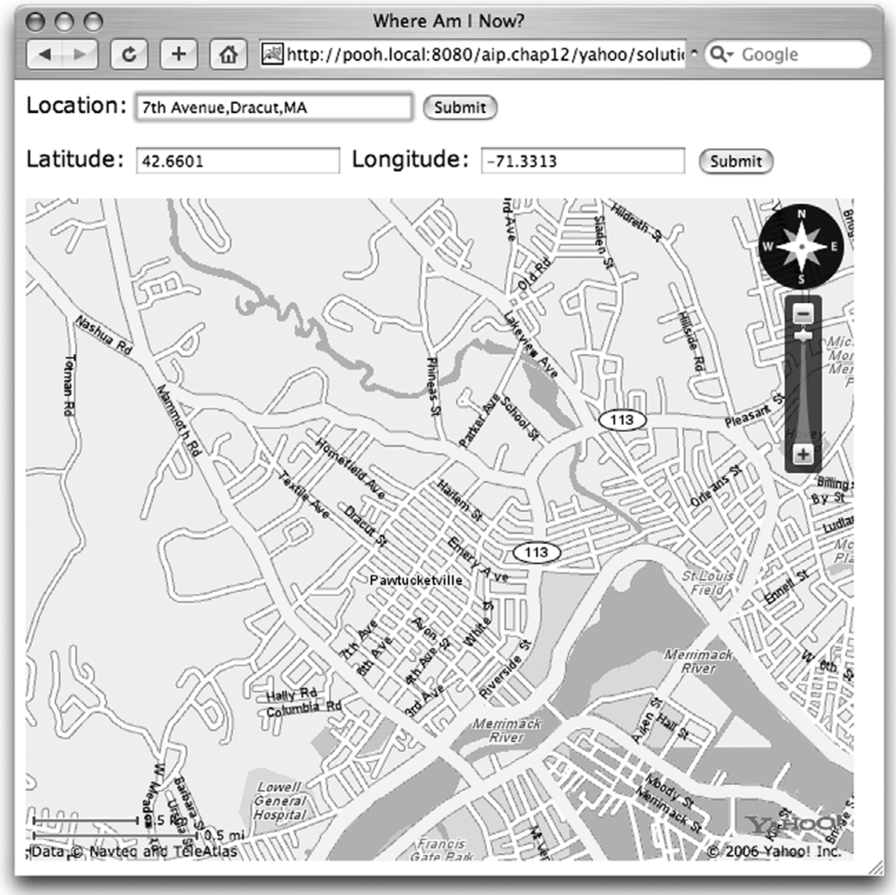

Figure 12.4 Revised page with address location

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 12.4 Yahoo! Maps with Geocoding page

```
<html>
  <head>
    <title>Where Am I Now?</title>
    <script type="text/javascript"
      src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="prototype-1.5.1.js"></script>
    <script type="text/javascript">
      var map;

      window.onload = function() {
        map = new YMap($('theMap'));
        map.addPanControl();
        map.addZoomLong();
      };

      function showMap() {
        var zoomLevel = 4;
        var latitude = $F('latitude');
        var longitude = $F('longitude');
        var point = new YGeoPoint(latitude,longitude);
        map.drawZoomAndCenter(point,zoomLevel);
      }

      function findLocation() {
        new Ajax.Request(
          '/aip.chap12/proxy',
          {
            method: 'get',
            parameters: {
              '.serviceUrl.':
                'http://api.local.yahoo.com/MapsService/V1/geocode',
              appid: 'org.bibeault.aip',
              location: $F('locationField')
            },
            onSuccess: onCoordsObtained
          }
        );
      }

      function onCoordsObtained(request) {
        var xml = request.responseXML;
        document.mapForm.latitude.value =
          xml.getElementsByTagName('Latitude').item(0)
          .firstChild.data;
        document.mapForm.longitude.value =
          xml.getElementsByTagName('Longitude').item(0)
          .firstChild.data;
        showMap();
      }
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    </script>
  </head>
  <body>
    <div>
      <form name="geocodeForm"
            onsubmit="findLocation();return false;">
        Location: <input type="text" id="locationField"
                         name="location" style="width:200px;"/>
        <input type="submit"/>
      </form>
    </div>

    <div>
      <form name="mapForm" onsubmit="showMap();return false;">
        Latitude: <input type="text" name="latitude"/>
        Longitude: <input type="text" name="longitude"/>
        <input type="submit"/>
      </form>
    </div>
    <div id="theMap" style="width:600px;height:480px;"></div>
  </body>
</html>
```

12.1.4 *Yahoo! Traffic*

As intrepid storm chasers, it behooves us to be aware of situations on the roads that we are traveling on (with maps courtesy of our previous examples), so we want to make one more addition to our Yahoo! Maps page before setting out to videotape some incredible storm footage.

Problem

We want to be aware of traffic incidents in the area that we have mapped using the page we created in the previous solutions.

Solution

The Yahoo! Traffic API (http://developer.yahoo.com/traffic/index.html) is the perfect solution for this problem (assuming that we are storm chasing within the United States). Like the Yahoo! Maps Geocoding API, the Traffic API is a REST service, so we can employ many of the same techniques that we used in the previous solution to convert address locations to latitude and longitude coordinates. The URL for this service is http://api.local.yahoo.com/MapsService/V1/trafficData.

Once again we'll send as parameters the required application key that we registered with Yahoo!, as well as the latitude and longitude that we are interested in obtaining information about. Similar to the Geocoding API, the traffic service

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

accepts many other parameters, but all we need for our purposes are the latitude and longitude of the current map.

We're going to code our page such that once a map has been drawn, we can query the service for any traffic incidents that are reported for that area. So, we'll need a button that we'll add at the bottom of the map display area:

```
<div>
  <form name="trafficForm" onsubmit="showTraffic();return false;">
      <input type="submit" id="trafficButton"
             value="Show Traffic Alerts" disabled="disabled"/>
  </form>
</div>
```

Note that because we don't want the button to be available until *after* a map has been displayed, we initially disable it. At the end of our `showMap()` function, we add code to enable it once the map has been shown:

```
$('trafficButton').disabled = false;
```

Upon the click of the enabled traffic button, the `showTraffic()` function performs what should be a familiar-looking operation:

```
function showTraffic() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://api.local.yahoo.com/MapsService/V1/trafficData',
        appid: 'org.bibeault.aip',
        latitude: $F('latitudeField'),
        longitude: $F('longitudeField')
      },
      onSuccess: onTrafficObtained
    }
  );
}
```

This function makes a request to the Traffic API in the same manner as we made the request to the Geocoding API, differing only in the details provided: providing the service URL for the Traffic API, passing the latitude and longitude values as query parameters, and specifying a different callback handler.

The callback function for this request is quite a bit different from the one we coded for the Geocoding service. In the callback for that solution, we simply digested the return values and shoved them into the form fields for the latitude and longitude values.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

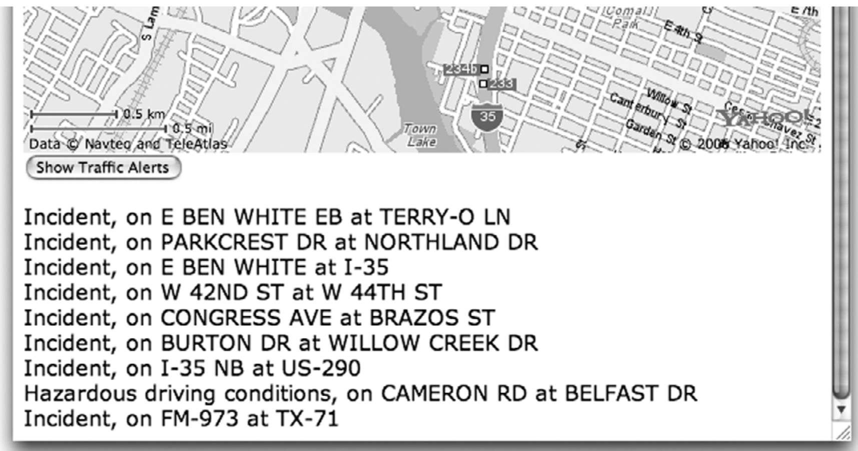

Figure 12.5 Traffic conditions added

For the traffic response, we want to display any incidents returned directly on the page rather than in form elements. Two methods are frequently used to accomplish this:

- We can use the DOM manipulation API to create the element nodes necessary to display our data as we see fit.
- We can build up some markup in a string buffer and place it into an element via its `innerHTML` property.

We'll see the first technique in section 12.3.1, so in this one we'll use the `innerHTML` mechanism. The result of all these additions is shown in figure 12.5.

In either scenario, we need an element in which to place the new elements, so below the traffic button in the page body we add

```
<div id="trafficAlerts"></div>
```

Then, we can code our callback function as follows:

```
function onTrafficObtained(request) {
  var xml = request.responseXML;
  var results =
    $A(xml.getElementsByTagName('Result'));
  $('trafficAlerts').innerHTML = '';
  if (results.length > 0) {
    results.each(
      function(result) {
        $('trafficAlerts').innerHTML +=
```

❶ Obtains <Result> children

❷ Collects <Title> children of node, emits info

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
            result.getElementsByTagName('Title').item(0)
            .firstChild.data +
            '<br/>';
        }
      )
    }
    else {
      $('trafficAlerts').innerHTML = 'No incidents reported';
    }
  }
```

The XML response for this service consists of a <ResultSet> element that contains a list of <Result> elements, each of which in turn contains information for a reported traffic incident. The child elements include information on the incident, even the coordinates. But since we're only interested in a synopsis, we'll grab the contents of the <Title> child element that is present in each <Result>.

A trimmed version of a response document from the Traffic API is shown here:

```
<ResultSet xsi:schemaLocation="urn:yahoo:maps
http://api.local.yahoo.com/MapsService/V1/TrafficDataResponse.xsd">
  <LastUpdateDate>1144350730</LastUpdateDate>

  <Result type="incident">
        <Title>Incident, on E OLTORF ST at BENJAMIN ST</Title>
        <Description>COLLISION PRIVATE PROPERTY J</Description>
        <Latitude>30.231270</Latitude>
        <Longitude>-97.734753</Longitude>
        <Direction>N/A</Direction>
        <Severity>3</Severity>
        <ReportDate>1144350420</ReportDate>
        <UpdateDate>1144350705</UpdateDate>
        <EndDate>1144352505</EndDate>
  </Result>

/* more Result elements removed to save space */

</ResultSet>
```

The handler operates by obtaining all <Result> children of the document ❶ (conveniently converting the node list into an array using Prototype's `$A()` function), finds the <Title> children of those nodes ❷, and uses the content of those elements to format the content of the trafficAlerts <div> on our page.

Styling the output to make it completely beautiful is probably something you'd want to do before too much longer.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

With the help of the Yahoo! Traffic REST API, we have added the ability to make ourselves aware of any traffic conditions in our area that could affect our ability to quickly move about—something rather important if a storm ends up chasing *us*.

This solution uses mechanisms similar to that used in the previous Geocoding solution, but rather than filling in form elements with the returned information, we used the `innerHTML` property of a `<div>` element to dynamically add text directly to our page. While fairly straightforward and easy, this mechanism isn't always the best choice.

The markup we generated is very simple: text strings delimited with `<br/>` tags. If we elect to attempt to add more complex markup to embellish the style and appearance of our information, we'd quickly discover that creating and maintaining markup in text strings is less than pleasant.

Another mechanism to dynamically add elements to our page under client-side control is to use the DOM API to directly add elements to the HTML DOM tree for our page. Although that involves a lot more code than we saw with the `innerHTML` mechanism, it's not too arduous and is certainly notationally cleaner than building markup in strings. We'll visit that technique in a later section. The complete code for our example, with additions and changes in bold, is shown in listing 12.5 and can be found in the downloadable source code for this chapter.

Listing 12.5 Revised page with traffic alerts

```
<html>
  <head>
    <title>What's Going On?</title>
    <script type="text/javascript"
    src="http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="prototype-1.5.1.js"></script>
    <script type="text/javascript">
      var map;

      window.onload = function() {
        map = new YMap($('theMap'));
        map.addPanControl();
        map.addZoomLong();
      };

      function showMap() {
        var zoomLevel = 4;
        var latitude = $F('latitudeField');
        var longitude = $F('longitudeField');
        var point = new YGeoPoint(latitude,longitude);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  map.drawZoomAndCenter(point,zoomLevel);
  $('trafficButton').disabled = false;
}

function findLocation() {
  new Ajax.Request(
    '/aip.chap12proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://api.local.yahoo.com/MapsService/V1/geocode',
        appid: 'org.bibeault.aip',
        location: $F('locationField')
      },
      onSuccess: onCoordsObtained
    }
  );
}

function onCoordsObtained(request) {
  var xml = request.responseXML;
  document.mapForm.latitude.value =
    xml.getElementsByTagName('Latitude').item(0).
    firstChild.data;
  document.mapForm.longitude.value =
    xml.getElementsByTagName('Longitude').item(0).
    firstChild.data;
  showMap();
}

function showTraffic() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://api.local.yahoo.com/' +
          'MapsService/V1/trafficData',
        appid: 'org.bibeault.aip',
        latitude: $F('latitudeField'),
        longitude: $F('longitudeField')
      },
      onSuccess: onTrafficObtained
    }
  );
}

function onTrafficObtained(request) {
  var xml = request.responseXML;
  var results = $A(xml.getElementsByTagName('Result'));
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          $('trafficAlerts').innerHTML = '';
          if (results.length > 0) {
            results.each(
              function(result) {
                $('trafficAlerts').innerHTML +=
                  result.getElementsByTagName
                    ➥ ('Title').item(0).firstChild.data +
                  '<br/>';
              }
            )
          }
          else {
            $('trafficAlerts').innerHTML = 'No incidents reported';
          }
        }
      </script>
    </head>
    <body>
      <div>
        <form name="geoForm" onsubmit="findLocation();return false;">
          Location: <input type="text" id="locationField"
                           name="location" style="width:200px;"/>
          <input type="submit"/>
        </form>
      </div>

      <div>
        <form name="mapForm" onsubmit="showMap();return false;">
          Latitude: <input type="text" id="latitudeField"
                           name="latitude"/>
          Longitude: <input type="text" id="longitudeField"
                            name="longitude"/>
          <input type="submit"/>
        </form>
      </div>

      <div id="theMap" style="width:600px;height:480px;"></div>

      <div>
        <form name="trafficForm"
              onsubmit="showTraffic();return false;">
            <input type="submit" id="trafficButton"
                   value="Show Traffic Alerts" disabled="disabled"/>
        </form>
      </div>

      <div id="trafficAlerts"></div>

    </body>

  </html>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

With that, we'll leave the rest (pun absolutely intended) of the Yahoo! APIs for you to explore on your own. What other cool features could you add to our map page?

Now let's turn our attention to another well-known web celebrity.

12.2 The Google Search API

Few people who have ever searched for anything on the Internet would be unfamiliar with Google as one of the Web's preeminent search engines. In this section we'll see how the public API that Google provides can be used to instrument search capability into our own pages.

12.2.1 Google search

Many of us maintain blogs or other websites with narrative sections. Frequently within those passages of text, we introduce terms and concepts that might be unfamiliar to the readers of our sites.

Let's suppose, once again donning our storm chaser persona, we are writing about weather conditions. Within the text of our passages we might use terms like *mammatus*, *rear-flank downdraft*, *dry line*, *mesocyclone*, or *mesoscale convection*, which may be familiar concepts to other weather aficionados but are not exactly everyday concepts to most people.

Problem

In our blogs or other online text postings, we'd like to build in the ability for readers to instantly search for terms that they might not be familiar with. We want to do so without forcing them to open another browser window, and we certainly don't want them navigating away from our pages. We've worked hard to get them there in the first place; the last thing we want to do is send them away!

Solution

Before we can instrument our pages with search capabilities, we need to set up a means to perform the search. Building our own search engine is obviously out of the question, so we'll leverage the Java Google API to perform the search on our (or rather, our reader's) behalf.

As with Yahoo!, Google also makes other APIs available, including an old-fashioned SOAP API. (The rumor mill has it that Google, the company, has been striving to get away from SOAP-based web services in order to embrace RESTful web services because they are more scalable and easy to use.) But in order to present a variety of API types, we'll be using the Java API in this section.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Once we have a Java class that can search for us, it will be an easy matter to create a servlet that we can use as an Ajax target on our pages to retrieve search results for whatever term we wish.

The simple search engine

The Google Search API is surprisingly simple. There are only four classes that we need be concerned with: GoogleSearch, GoogleSearchResult, GoogleSearchResultElement, and the exception class GoogleSearchFault, which is thrown in the event of a problem.

But to begin, we must obtain these Google Search classes, and we must also sign up for a Google license key in a similar fashion as was required to use the Yahoo! Services. Unlike the Yahoo! application key, which we got to choose, Google will assign you a key that they generate on your behalf. You can find instructions on downloading the development kit and obtaining a license key at www.google.com/apis/.

In the development kit you'll find a file named `googleapi.jar`. Place this JAR file in the `WEB-INF/lib` folder of your web application, and be sure that it is in the class path of your compile-time build.

With that simple setup in place, we're ready to write a simple, reusable class to perform searches using the Google engine. When we execute this program with an input term of, say, *outflow boundary*, the result will be along these lines (truncated to conserve space)

```
title: <b>Outflow</b> Technologies
url: http://www.outflow.net/
snippet: Offers web design, web marketing, web hosting, and
e-commerce services.

title: <b>Outflow</b>: winds flowing outward from thunderstorms
url: http://ww2010.atmos.uiuc.edu/(Gh)/guides/mtr/svr/dngr/oflow.
rxml
snippet: <b>Outflow</b>. winds flowing outward from thunderstorms.
 Thunderstorm winds also cause<br>  widespread damage and occasion
al fatalities. Thunderstorm &quot;straight-line&quot; <b>...</b>

title: <b>Outflow</b> Phenomena: downbursts
url: http://ww2010.atmos.uiuc.edu/(Gh)/guides/mtr/svr/comp/out/
home.rxml
snippet: <b>Outflow</b> Phenomena. downbursts. This section is on
visual identification of<br>  macrobursts, microbursts, gust front
s and other <b>outflow</b> phenomena. <b>...</b>

... more ...
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Well, the first result was a bit disappointingly inappropriate, but the remaining results are right on target.

Let's set up this SimpleGoogleSearch class as shown listing 12.6.

Listing 12.6 The SimpleGoogleSearch class

```
package org.bibeault.aip.search.google;

import com.google.soap.search.*;                    ❶ Imports Google Search API

public class SimpleGoogleSearch {

    private GoogleSearch googleSearch;              ❷ Records Google search engine

    public SimpleGoogleSearch(String clientKey) {   ❸ Constructs simple search class
        this.googleSearch = new GoogleSearch();
        this.googleSearch.setKey(clientKey);
    }

   public GoogleSearchResultElement[]               ❹ Executes search
      search(String searchTerm)
            throws GoogleSearchFault {
        this.googleSearch.setQueryString(searchTerm);
        GoogleSearchResult googleSearchResult =
            this.googleSearch.doSearch();
        return googleSearchResult.getResultElements();
    }
}
```

Not as many lines of code as you expected, is it?

After importing the Google Search API ❶, we declare an instance variable to hold the Google search engine instance that this class will wrap ❷. The constructor ❸ for our simple search class accepts a Google license key (obtained from the Google site). This key is an unintelligible string of characters that is impossible to remember (at least for us humans) and is generated by Google when you sign up. Within the constructor we create and store an instance of the Google search engine and give it the passed access key.

After construction, our class will be ready to perform searches. We designed the class in this fashion because the same instance of the Google search engine can be used to perform many searches. So we instantiate it once at construction time and use it whenever a search is performed.

Those searches will be executed by the `search()` method ❹ by passing it a search term string, which has similar semantics to a string typed into the Google website. The search is conducted by setting the passed search term into the search

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

engine instance and calling its `doSearch()` method, which returns an instance of a GoogleSearchResult, a container for the results of the search. As the return value of the method, the array of results stored in that container, as instances of GoogleSearchResultElement, are fetched. Should anything go awry, an instance of GoogleSearchFault is thrown.

As with the content grabber, we add a `main()` function that can be used to perform a minimal test of the class as follows:

```
public static void main(String[] args) throws Exception {
    if (args.length == 0)
      throw new Exception("A search term must be provided");
    SimpleGoogleSearch searcher =
      new SimpleGoogleSearch("aApewexQFHItVSrlTMDk2iglRbhB+6AR");
    GoogleSearchResultElement[] results = searcher.search(args[0]);
    for (GoogleSearchResultElement result : results) {
      System.out.println("title: " + result.getTitle());
      System.out.println("url: " + result.getURL());
      System.out.println("snippet: " + result.getSnippet() + '\n');
    }
  }
```

Go ahead and compile the class and run it as a program as described previously. The first parameter on the command line will be used as the search term.

Of course, the GoogleSearch class has many more options than we exposed in our very simple search class. For example, you can specify the number and offset of the returned results to facilitate paging of a long results list, or even set up language restrictions. Explore the API of the GoogleSearch object for details.

The simple search servlet

Now that we're armed with the SimpleGoogleSearch class to do the lion's share of the work necessary to perform the search, writing a servlet to obtain the results and return them as a response (to an Ajax request, in our case) should be a simple matter.

Indeed, we'll discover that the search portion of the servlet is almost trivial. What's going to consume the majority of the servlet code, and our time in developing it, concerns the matter of the format of the returned data.

In the previous solutions in this chapter, we return an XML document as the response to our Ajax calls as that was what we were given from the APIs we employed. In this case, however, *we* are generating the response ourselves from the information that we'll garner from the GoogleSearchResultElement instances.

We *could* create XML and serialize it to the response stream, only to digest it on the client. But in this case, as we know that we'll be digesting the result of our

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

search request in JavaScript code, we'll format the response using JSON, which is much simpler to deal with on the client side than XML.

It could be argued that using JSON limits the use of our servlet to JavaScript environments. We know that our own use will be limited to JavaScript, and that this limitation won't be a factor for us. Should we want our search servlet to be more widely reusable, we could revisit that decision.

Our response will be a list of results, each with three fields: a title, a URL, and a text snippet. In JSON notation, a three-result response with bogus data might be

```
[
 { title: 'Title 1', url: 'http://url1/', snippet: 'Snippet 1' },
 { title: 'Title 2', url: 'http://url2/', snippet: 'Snippet 2' },
 { title: 'Title 3', url: 'http://url3/', snippet: 'Snippet 3' }
]
```

The square brackets denote an array, the braces each delimit an object element of the array, and the label/value pairs denote the properties and corresponding values for the objects. Given that, and armed with our simple search class, our search servlet is shown in listing 12.7.

Listing 12.7 The search servlet

```
package org.bibeault.aip.search.google;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import com.google.soap.search.*;

public class SimpleGoogleSearchServlet extends HttpServlet {

  public static final String
    KEY_SEARCH_TERM = "term";

  protected void doGet(
    HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
    String searchTerm = request.getParameter(KEY_SEARCH_TERM);
    SimpleGoogleSearch searcher =
      new SimpleGoogleSearch("aApewexQFHItVSrlTMDk2iglRbhB+6AR");
    try {
      GoogleSearchResultElement[] results =
        searcher.search(searchTerm);
      StringBuilder responseBody = new StringBuilder();
      responseBody.append('[');
      for (GoogleSearchResultElement result : results)
        appendResultAsJSON(responseBody, result);
```

❶ Expects term parameter

❷ Obtains term parameter

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        responseBody.append(']');
        response.setContentType("text/plain");
        response.setContentLength(responseBody.length());
        response.getWriter().print(responseBody.toString());
      } catch (GoogleSearchFault e) {
        e.printStackTrace();
        throw new ServletException("Search error: " + e, e);
      }
    }

    private void appendResultAsJSON(                    ❸ Formats JSON response
        StringBuilder responseBody,
       GoogleSearchResultElement result) {
      responseBody
        .append('{')
        .append("title:'")
        .append(escapeQuotes(result.getTitle())).append("',")
        .append("snippet:'")
        .append(escapeQuotes(result.getSnippet())).append("',")
        .append("url:'").append(escapeQuotes(result.getURL()))
        .append("'")
        .append("},");
    }

    private String escapeQuotes(String text) {
      return text.replaceAll("'", "\'");
    }

  }
```

We've coded our servlet to expect a single request parameter named `term` that will contain the search term ❶. Upon receiving a GET request ❷ (POST could also easily be supported), the `term` parameter is obtained. (Some error checking here would be nice.)

An instance of SimpleGoogleSearch is created using a Google license key. Normally, something such as this license key should never be hard-coded into the code in this manner. It should be provided via an external resource—perhaps in a properties file, or as a deployment descriptor context parameter.

Using this search engine instance, the search is performed and the results obtained.

To format the JSON response to send back to the client, a StringBuilder instance is created and the response body is generated with the help of the `appendResultsAsJSON()` implementation method ❸. Note that another implementation method, `escapeQuotes()`, is employed to ensure that single quotes in the returned text won't foul up our JSON syntax.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The response is configured with a MIME type of plain text and the length of the JSON response body. Finally, the JSON results are streamed to the response output.

Adding the servlet and its mapping to the `web.xml` deployment descriptor of our web application is accomplished with the following:

```
<servlet>
  <servlet-name>SimpleGoogleSearchServlet</servlet-name>
  <servlet-class>
    org.bibeault.aip.search.google.SimpleGoogleSearchServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>SimpleGoogleSearchServlet</servlet-name>
  <url-pattern>/search</url-pattern>
</servlet-mapping>
```

This sets us up to begin writing the pages that will make use of our new search facility.

Instrumenting pages for search

Getting back to our original intent, we want to instrument our text passages so that users can easily search on unfamiliar terms with a single click. We could make these terms, as they appear in our text, anchor tags, and use JavaScript to trigger the Ajax request for the search results. But that would make it difficult to apply individual styling to differentiate our “automatic search terms” from “real” links. Besides, that would also be kind of wordy in the page code.

Instead, let’s be a little cleverer and hijack the use of the `<abbr>` tag, a little-used HTML/XHTML tag that is originally intended to highlight abbreviations in HTML pages. With a lot of tongue-in-cheek rationalization, we can justify our use of this tag by asserting that the search term is just an “abbreviation” for its results. But seriously, if it bothers you to hijack the `<abbr>` tag, you can also use a `<span>` with a style class applied. It can work the same way; it’s just a bit wordier.

Examine the following text snippet from a page employing the `<abbr>` tag in this manner:

```
<p>
  I find that the accumulation of
  <abbr onclick="searchFor(this);">stratocumulus</abbr> and
  absence of <abbr onclick="searchFor(this);">solar
  radiation</abbr> presages an increased rate of
  <abbr onclick="searchFor(this);">condensation</abbr>. We may
  expect vertical <abbr onclick="searchFor(this);">precipitation
  </abbr> of moisture at any time.
</p>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Note that we have embedded each term that we want the reader to be able to search for in an `<abbr>` tag whose `onclick` event handler invokes a JavaScript function that passes the `<abbr>` element as its parameter. That's pretty easy, but it *is* a bit messy, and it does seem redundant to have to put the same `onclick` clause on each and every term that we want to instrument.

So let's automate it by writing a JavaScript class that can handle all of this on behalf of our pages. We've seen JavaScript classes throughout this book, and creating them was extensively covered in chapter 3, so after presenting the code for this call in listing 12.8, we'll gloss over all but the interesting aspects of the class.

Listing 12.8 The SearchInstrumenter class

```
SearchInstrumenter = Class.create();

SearchInstrumenter.prototype = {

  DEFAULT_ELEMENT_TYPE: 'abbr',
  DEFAULT_RESULTS_CONTAINER: 'resultsContainer',
  SEARCH_URL: '/aip.chap12/search',

  initialize: function(options) {     <-- 1 Instruments each element of specified type
    this.options = Object.extend(
      {
        elementType: this.DEFAULT_ELEMENT_TYPE,     <-- 2 Supplies default options
        resultsContainer: this.DEFAULT_RESULTS_CONTAINER
      },
      options
    );
    var self = this;
    $$ (this.options.elementType)
      .each(
        function(element) {     <-- 3 Assigns onclick handler to each element
          element.onclick = function() {
            self.doSearch(element.innerHTML);
          }
        }
      );
  },

  doSearch: function(term) {     <-- 4 Performs search upon click
    new Ajax.Request(
      this.SEARCH_URL,
      {
        method: 'get',
        parameters: {term: term},
        onSuccess: this.showResults.bind(this),
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        onFailure: this.showError.bind(this)
      }
    );
    $(this.options.resultsContainer).innerHTML =
      'Searching for ' + term + '...';
  },

  showResults: function(request) {        ⑤ Obtains and displays response text
    var jsonResponse = request.responseText;
    eval ('(results='+jsonResponse+')');
    $('resultsContainer').innerHTML = '';
    results.each(
      function(result) {
        $('resultsContainer').innerHTML +=
          '<p>Title: <b>' + result.title + '</b><br/>' +
          'Summary: ' + result.summary + '<br/>' +
          'URL: ' + result.url +'</p>';
      }
    );
  },

  showError: function(request) {        ⑥ Displays error message
    $(this.options.resultsContainer).innerHTML =
      request.responseText;
  }

}
```

The constructor for this class ❶ will instrument every instance of a specific type of tag in our page. It accepts a single `options` parameter that allows us to override the default values provided by the class. The options for this class ❷ are `elementType` and `resultsContainer`, which have defaults of `abbr` and `resultsContainer`, respectively.

Once the element type to instrument is determined, all instances of that element are located and an `onclick` handler is assigned to each ❸. This frees us from having to add the handler by hand on each and every element to be instrumented, as we were required to in the example text shown earlier.

The `onclick` event handler assigned to each located element is the `doSearch()` method defined by this class. Note the use of the function's closure to make the class instance and the element reference available when this handler triggers. (If all that sounded like gobbledegook, review the concept of *closures* in chapter 3.)

The `doSearch()` method used as the `onclick` handler ❹ makes an Ajax request to our search servlet using techniques that we've seen in previous solutions in this chapter. Of particular note is the use of an object hash to pass the

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

request parameter, and the assignment of other class methods as the success and failure handlers for the request.

The `onSuccess` event handler `showResults()` ❺ obtains the response text from the XHR instance. As you'll recall, the search servlet returns a JSON construct containing the search results. We evaluate that response and iterate over the results, format some HTML containing the data retrieved, and set it as the content of the element recorded as the results container.

The final part of our class is the failure handler `showError()` ❻. It obtains the error message text from the request instance and puts it into the results container for all to see.

Testing the search instrumenter

Finally, we're ready to write a page that uses our handy search capabilities. It can be found in the downloadable source code for this chapter, and is shown in listing 12.9. This unimaginative page places the search results in a block under the text, as shown in figure 12.6.

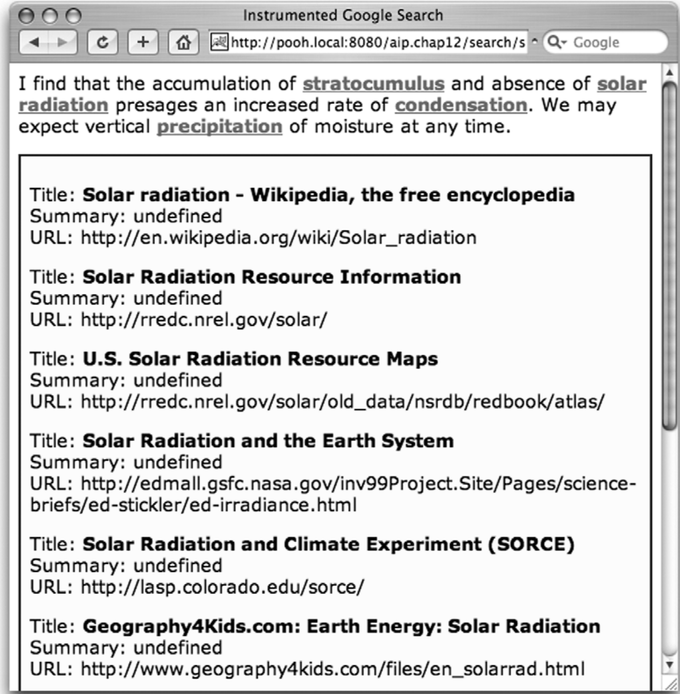

Figure 12.6
All about solar radiation!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 12.9 What's a stratocumulus?

```
<html>
  <head>
    <title>Instrumented Google Search</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="SearchInstrumenter.js">
    </script>
    <script type="text/javascript">
      window.onload = function () {
        new SearchInstrumenter();
      };
    </script>
    <style type="text/css">
      abbr {
        color: green;
        font-weight: bold;
        text-decoration: underline;
        cursor: pointer;
      }
      #resultsContainer {
        border: 2px ridge maroon;
        background-color: #ffffcc;
        padding: 8px;
      }
    </style>
  </head>

  <body>
    <div>
      <p>
        I find that the accumulation of <abbr>stratocumulus</abbr>
        and absence of <abbr>solar radiation</abbr> presages an
        increased rate of <abbr>condensation</abbr>. We may expect
        vertical <abbr>precipitation</abbr> of moisture at any
        time.
      </p>
    </div>
    <div id="resultsContainer"></div>
  </body>
</html>
```

This deceptively simple page makes use of all the technology that we've set up in this section to allow visitors to our page to perform an automatic search on any term that we've indicated with the `<abbr>` tag by simply clicking it.

With just a little more CSS and JavaScript magic, we could do something much more creative. For example, we could make the result appear in a floating

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

`<div>` near the original search term. Or we could expand the results element inline within the text until dismissed. The possibilities are limited only by your imagination.

Discussion

With this solution, we achieved the ability to initiate asynchronous searches on any search terms we desire. We developed a way to gather search terms directly from the HTML text, and we explored a means of returning dynamic data to the page in a format other than XML.

You could argue that the use of JSON rather than XML to return the data might limit our search service to environments capable of consuming JSON. But in a servlet environment, it is unlikely that anything other than an HTML page would be the consumer of our data. Besides, we could always write a JSON-to-XML adapter if need be.

As with the other solutions in this chapter, this example could use a hefty dose of error checking and recovery. In fact, while testing it was discovered that the Google service returns a "service is temporarily unavailable" error for a nontrivial percentage of the requests made to the service. How would you improve our search components to handle that?

Oh, and the URLs in the results? They should probably be links, don't you think?

12.3 Flickr photos

It's all about sharing the photos.

Whether you're a hard-core storm chaser with images of that magnificent super-cell thunderstorm, a shutterbug with tons of photos of Texas Spring wildflowers, or a doting grandparent with just-too-cute pictures of the grandkids, the whole point of photographs is being able to share them.

One of the fastest-growing photo-sharing sites on the Web is Flickr (www.flickr.com), and to our delight it exposes a deep and broad API for developers in a multitude of different forms. API kits for everything from Java, .NET, Delphi, and Ruby through Perl are available, as well as request-based APIs for SOAP, XML-RPC, and REST. As before, we'll focus on the REST interface, as that is one of the most appropriate for use from Ajax-enabled pages.

Similar to other web APIs, Flickr requires you to register and receive an API key that is used to identify the originator of each request. This API key consists of a random string of letters and numbers and is easy to obtain by visiting www.flickr.com/services/api/misc.api_keys.html.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Once obtained, we can use it to make requests to the Flickr REST API. All REST requests to the Flickr service use the base URL www.flickr.com/services/rest/.

Each request must provide two required query parameters: `api_key`, which supplies the API key identifying us to the system, and `method`, which identifies the function that we wish the service to perform on our behalf. *Flickr methods* are not to be confused with *class methods*; they are just text strings that are passed to the Flickr service to identify the operation to be carried out. Depending on the method identified, there may be more parameters—some required, some optional.

Flickr exhibits the concept of public and private photo collections. Either is available via the web API, but dealing with private collections, not surprisingly, requires a rather complex authentication protocol to be employed. For our purposes here, we'll avoid all that by focusing on public collections.

12.3.1 Flickr identification

In this section we'll develop a page that will display the thumbnails of a friend's public photo collection. Each thumbnail can be expanded to a larger version when clicked on.

You might ask, "Why bother?" After all, we can just statically code image tags with the URLs to the images, can't we?

We sure could. Flickr doesn't seem to care if you embed references to their hosted photos in your own pages. But isn't it a bit dated to hand-code references in static pages to external resources that could change without our knowing about it? References that we'll need to check often and adjust as they come and go? What a bother!

Instead, by relying on the Flickr APIs, we obtain real-time information about available resources and, with our Ajax and DHTML skills, dynamically create a page that is never out of date.

Problem

We want to construct a page of active thumbnail images for the public photo collection of a friend. We know the friend's Flickr username, but as soon as we look into the API for the method that returns a list of the friend's public photos, we discover a problem. It's not the friend's username that we need, but his *NSID*—an internal identifier that is used in most Flicker public methods to identify the target user.

This should not be confused with the API key. The API key identifies who is *making* the request. The NSID identifies whose account is being *targeted* by the request.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

So before we look into a solution for creating our thumbnails page, we need to whip one up to obtain a user's NSID given his or her username.

Solution

Figure 12.7 shows you the finished result of the code featured in this solution.

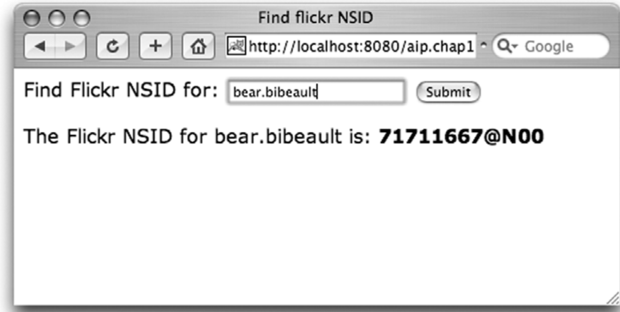

Figure 12.7 Finding a Flickr NSID

The Flickr *method* to obtain an NSID given a username is

```
flickr.people.findByUsername
```

All the Flickr methods begin with the string `flickr`, which is then followed by a category name (people, contacts, groups, and so on), and finally the specific service being requested.

As we want to obtain the NSID given a username, we'll begin our page by creating a simple form to accept the username string and trigger the request:

```
<form name="queryForm" onsubmit="findNSID();return false;">
  Find Flickr NSID for:
  <input type="text" name="username"/>
  <input type="submit"/>
</form>
```

When the user types in a username and clicks the submit button, the `findNSID()` function is called:

```
function findNSID() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
      method: 'get',
      parameters: {
        '.serviceUrl.':
          'http://www.flickr.com/services/rest/',
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
          api_key: '78eaa5287f3f0b37dfba77ef40c7df03',
          method: 'flickr.people.findByUsername',
          username: $F('username')
        },
        onSuccess: onResultObtained
      }
    );
    $('resultContainer').innerHTML = '';
  }
```

You should find the structure of this function quite familiar at this point. An Ajax request is made through our proxy relay agent on the server specifying our Flickr API key, the Flickr method to be invoked (not to be confused with the HTTP method supplied to Prototype), and the username to be looked up.

The `onSuccess` event handler for this request is

```
function onResultObtained(request) {
  var xml = request.responseXML;
  if (xml.getElementsByTagName('rsp')[0].getAttribute('stat')
        == 'ok') {
    showResults(xml);
  }
  else {
    $('resultContainer').innerHTML = 'Request failed.';
  }
}
```

The response document for all requests consists of an `<rsp>` element, which in turn contains the information for the response as child elements. An attribute on the `<rsp>` element named `stat` will specify the value `ok` or `fail`, depending on whether the request succeeded.

A failed request would look something like this:

```
<rsp stat="fail">
  <err code="112" msg="Method &quot;&quot; not found" />
</rsp>
```

Our handler function tests this attribute to determine whether the request has failed or succeeded. If a failure is detected, the results container `<div>` is set to a simple error message. (With a few more lines of JavaScript, we could be a lot friendlier and extract the specific error message from the response document.) If the request succeeded, the XML response document is passed to a function named `showResults()`.

In our previous solutions, we made liberal use of the `innerHTML` property to dynamically set the content of elements defined in order to display the results of our requests. Although that's an adequate means to accomplish this for simple

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

text or extremely simple markup, it doesn't scale very well. As soon as the markup to be generated gets a bit more complex or lengthy, building that markup in text strings becomes rather cumbersome and awkward—not to mention a maintenance headache in the making.

So in our `showResults()` function, we'll use a different means to dynamically create the content of our result container: DOM manipulation.

We've used the DOM API to query the content of XML documents that have been passed to us, but we can also use that same API to manipulate the DOM of the HTML document to our needs.

The format for a successful response from the Flickr service when the `findByUsername` method is used is along the lines of

```
<rsp stat="ok">
  <user id="71711667@N00" nsid="71711667@N00">
    <username>bear.bibeault</username>
  </user>
</rsp>
```

Given that, we code our `showResults()` function:

```
function showResults(xml) {
  var nsid = xml.getElementsByTagName('user')[0]
    .getAttribute('nsid');
  var p = document.createElement('p');
  p.appendChild(document.createTextNode('The Flickr NSID for ' +
              document.queryForm.username.value + ' is: '));
  var span = document.createElement('span');
  span.style.fontWeight = 'bold';
  span.appendChild(document.createTextNode(nsid));
  p.appendChild(span);
  $('resultContainer').appendChild(p);
}
```

As you'll recall, we pass the XML response document to the function, and we know that the check for successful requests has already been undertaken.

After we locate the NSID for the username and record it for later use, we're ready to start building new HTML elements in our document. First, a new paragraph element (`<p>`) is created. We want to place some text into that element, so we create some text and set it as a child of the paragraph element.

The NSID is to appear in a bold font, so it will be embedded in a `<span>` element with the appropriate style applied. The bolded `<span>` element is set as a child of the paragraph, and we finish up by adding the paragraph as a child of the results container.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Discussion

In this very simple solution, we gave ourselves the ability to obtain the NSID for any Flickr user given their username. This NSID is required to make any other method calls to the Flickr API in order to identify the target account.

This NSID is fixed and static for each username, so once obtained, it can be recorded and used directly if desired. We'll see this in our next solution.

We also explored a different means of populating a container element with dynamic results by using DOM manipulation rather than building markup in a string and setting it into the container through its `innerHTML` property.

Although this technique may at first seem like overkill or overly wordy compared to the `innerHTML` mechanism, for nontrivial markup it's easier to maintain in the long run and is more robust. One definite advantage that it has over `innerHTML` is that we don't run into quoting issues and other notational problems that inevitably arise when creating markup in text strings.

12.3.2 Flickr photos and thumbnails

Now that we have the NSID for the target account, we're ready for the fun part: getting photos from Flickr.

Problem

We want to construct a page of active thumbnail images for the public photo collection of a friend. Now that we know our friend's NSID, there's nothing stopping us.

When a thumbnail is clicked, we wish to display a larger-sized version of the photo. And we want to do all this without hard-coding anything except the NSID of the target account.

Solution

The Flickr method to obtain the list of public photos for a given NSID is

```
flickr.people.findPublicPhotos
```

In addition to the required method and API key parameters, this method requires a `user_id` parameter, which provides the NSID for the target account. Other optional parameters affect the type and amount of data returned, but we'll be keeping it simple for now.

An example of a successful response from this method is

```
<rsp stat="ok">
  <photos page="1" pages="1" perpage="100" total="3">
    <photo id="128217127" owner="71711667@N00" secret="09e814e0b0"
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
      server="51" title="DSC01660" ispublic="1" isfriend="0"
      isfamily="0" />
    <photo id="128217125" owner="71711667@N00" secret="ef6ad6886d"
      server="40" title="DSC01476" ispublic="1" isfriend="0"
      isfamily="0" />
    <photo id="128216010" owner="71711667@N00" secret="ff13ad56b9"
      server="46" title="DSC01118" ispublic="1" isfriend="0"
      isfamily="0" />
  </photos>
</rsp>
```

Believe it or not, this gives us everything we need in order to construct URLs to the photos on the Flickr servers.

Each photo URL served from Flickr is of the format

```
http://static.flickr.com/{server}/{id}_{secret}{suffix}.jpg
```

The `server`, `secret` and `id` values for each photo are taken directly from the attributes of the `<photo>` element in the response XML document, while the `suffix` specifies the size of the image as follows:

- _s: 75-by-75-pixel square
- _t: 100 pixels on longest side
- _m: 240 pixels on longest side
- *(none)*: 500 pixels on longest side
- _b: 1024 pixels on longest side
- _o: original image

If we wanted to create a URL for a thumbnail of the first photo in the example response shown earlier, substituting the `server`, `id`, `secret`, and appropriate `suffix` gives us a URL of

```
http://static.flickr.com/51/128217127_09e814e0b0_t.jpg
```

Using this URL as the source of an image element creates a thumbnail for the photo. By simply changing the suffix of the URL, say to `_b`, we get a URL to a bigger version of the same photo.

With that knowledge fresh in our minds, let's write our page. First, we want the thumbnails to automatically load when the page is displayed. So within the `<script>` element of the page header, we write the following:

```
window.onload = function() {
  new Ajax.Request(
    '/aip.chap12/proxy',
    {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        method: 'get',
        parameters: {
          '.serviceUrl.':
            'http://www.flickr.com/services/rest/',
          api_key: '78eaa5287f3f0b37dfba77ef40c7df03',
          method: 'flickr.people.getPublicPhotos',
          user_id: '71711667@N00'
        },
      onSuccess: onInfoObtained
      }
    );
  }
```

This handler kicks off the Ajax request to the `findPublicPhotos` method. Upon success, the handler for this request is

```
function onInfoObtained(request) {
  var xml = request.responseXML;
  if (xml.getElementsByTagName('rsp')[0].
          getAttribute('stat') == 'ok') {
    showThumbnails(xml);
  }
}
```

As with our other previous `onSuccess` handler, it checks the status of the Flickr method and if all is OK, the `showThumbnails()` function is called with the response XML document. Note that should anything go awry, nothing at all happens and the user will be presented with a blank page. Obviously, there's room for improvement here.

The `showThumbnails()` function is where all the really interesting things go on:

```
function showThumbnails(xml) {
  var photos = $A(xml.getElementsByTagName('photo'));
  photos.each(
    function(photo) {
      var baseUrl = 'http://static.flickr.com/' +
                    photo.getAttribute('server') + '/' +
                    photo.getAttribute('id') + '_' +
                    photo.getAttribute('secret');
      var thumbUrl = baseUrl + '_t.jpg';
      var photoUrl = baseUrl + '.jpg';
      var thumb = document.createElement('img');
      thumb.src = thumbUrl;
      thumb.style.cursor = 'pointer';
      thumb.onclick = showPhoto;
      thumb.photoUrl = photoUrl;
      $('thumbnailsContainer').appendChild(thumb);
    }
  );
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The function is called with the successful XML response document from which the list of <photo> elements is retrieved. As each <photo> element is iterated over, a base URL (minus the size suffix) is created from the information in the <photo> element. This base URL is then used to create URLs for the thumbnails and the full-sized photo by applying the _t suffix and empty suffix, respectively.

The DOM API is then used to create an <img> element that will show the thumbnail image. The cursor for the image element is set to pointer (the little hand) so that users will know that the thumbnail image can be clicked on, and the onclick handler for the image is set to the showPhoto() function.

The URL we created to show the medium-sized version of the photo is stored in a dynamic property directly on the image element named photoUrl for later retrieval. Remember, this is JavaScript where we can create our own properties on any object on the fly. That even applies to objects in the HTML DOM. How convenient.

Finally, the newly created <img> element is added as a child of a container <div> with the ID of thumbnailsContainer.

As a result of triggering this sequence of events, the thumbnails are loaded when the page is displayed, and each is armed and ready to display a larger version of the photo at a simple click. That operation occurs in the showPhoto() function that we declared as the onclick event handler of each thumbnail image:

```
function showPhoto() {
  var photo = $('photoElement');
  if (photo == null) {
    photo = document.createElement('img');
    photo.id = 'photoElement';
    $('photoContainer').appendChild(photo);
  }
  photo.src = this.photoUrl;
}
```

When the page first loads, we don't want a "broken image" icon to be displayed, so initially we don't even create an <img> element for the photo. Rather, on its first reference the nonexistence of the element is detected and we'll create the <img> element on the fly. So in our showPhoto() function we attempt to find the element with the ID of photoElement, and if it does not yet exist, it is created.

Once we have a reference to the element, be it newly created or not, we set the src attribute of the <img> element to the photo URL that was generated for the thumbnail image in the showThumbnails() function. This works because the context object (the object referenced by this) in the event handler is the object that triggered the handler—in this case, the thumbnail image.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

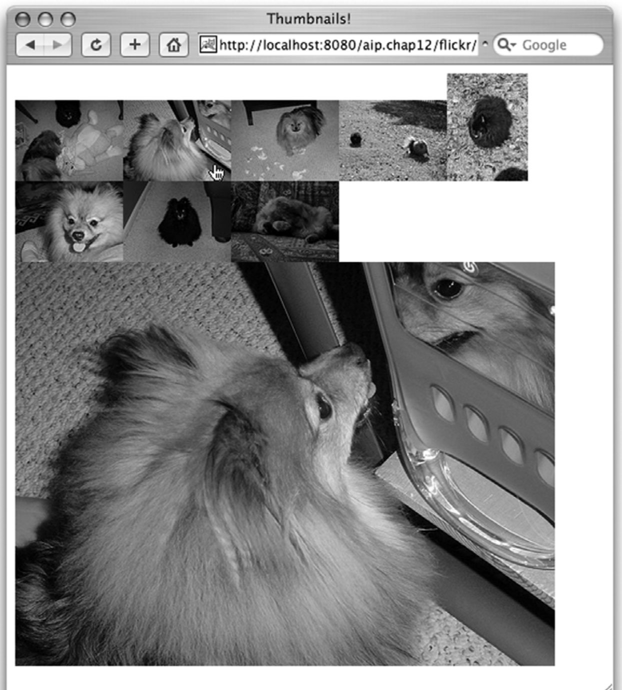

Figure 12.8
Who's that doggie in the window?

Finally, we're ready to write the body of the HTML page. But because we're pretty much generating everything dynamically, it's surprisingly simple:

```
<body>
  <div id="thumbnailsContainer"></div>
  <div id="photoContainer"></div>
</body>
```

The results of all that code (after page load and a click on one of the thumbnails) are shown in figure 12.8.

Discussion

In this section we learned how to use the Flickr REST API to dynamically access public photos and thumbnails from Flickr accounts—be it our own or anyone whose username we know.

It's quite obvious that the page we created could use some styling, and the usability leaves something to be desired in its initial, rather Spartan, state. But a good dose of CSS and a little JavaScript could improve the page in short order. The near-nonexistent error checking also needs to be rectified.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

In this solution, we hard-coded the NSID for the user whose account we wished to access. Since the NSID for a particular user never changes, this is a safe thing to do for users whose NSID we have already determined. If we did not wish to look up such NSID values in advance, we could combine this and the previous solution to deliver a one-two punch: submit an Ajax request to look up the NSID by username, and then in the success handler for that request, submit a second request that obtains the public photo list for the discovered NSID.

This page could (and probably should) also be enhanced to provide paging of long lists of photos using the optional arguments to the Flickr `getPublicPhotos` method.

We also saw a little JavaScript trick: tacking our own property onto an HTML DOM element for later reference. This is an extremely handy addition to our JavaScript toolkit when manipulating the DOM and creating dynamic pages.

12.4 But wait! As they say, there's more…

In this chapter we've looked at services exported by Yahoo!, Google, and Flickr to add some serious pizzazz to our dynamic web pages by using Ajax to access those services. But that's just the tip of the iceberg!

From A to Z, there are dozens, if not hundreds, of high-profile sites that provide APIs for us to use.

12.4.1 Amazon services

Amazon provides a robust set of services through which you can create your own applications, storefronts, or portals into the wealth of products offered by Amazon and its partners. Like many other services, Amazon exports a REST interface that makes it easy for us to use from Ajax-enabled web applications.

You can find details on signing up for and using the Amazon API at www.amazon.com/gp/aws/landing.html.

12.4.2 eBay services

Want to create the next great eBay application?

eBay offers a very broad and deep API to interface to eBay auctions and stores for everything from Java, to PHP, to SOAP, and yes, a REST API.

The REST API seems to be limited, at least at the time of this writing, to searching eBay listings. But we could also use any of the other APIs to create server-side agents for our Ajax-enabled pages should we wish to develop deeper eBay applications.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Information about the eBay Developer's Program can be found at http://developer.ebay.com/.

12.4.3 MapQuest

Not happy with the maps from Yahoo! or Google? Find out about the MapQuest OpenAPI at www.mapquest.com/features/main.adp?page=developer_tools_oapi.

12.4.4 NOAA/National Weather Service

Further interest in the weather? NOAA provides a SOAP-based API to forecasts, watches, and warnings. Check it out at www.nws.noaa.gov/forecasts/xml/.

12.4.5 More, more, more...

Search the Web and you'll find more web services than you could possibly know what to do with. A good place to start is the list at ProgrammableWeb at www.programmableweb.com/apilist.

Happy hunting!

12.5 Summary

In this chapter we've made some great strides. While we examined just a handful of the many web services available to us as developers, we explored a number of techniques that will be useful in dealing with just about any web service that may be available to us.

We learned how to use JavaScript APIs and how to integrate server-side APIs into our web applications. Perhaps most importantly, we learned how to circumvent the Ajax security sandbox in order to make requests to web services on servers other than our own.

Armed with this knowledge, there's virtually no end to the ways that we can utilize web services that various sites have generously made public for our use.

And we're not limited to using just a single service. In the next chapter we'll take a look at how we can combine services from different providers in a single web application.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13 Mashing it up with Ajax

This chapter covers

- Just what is a "mashup"?
- Deciding on data formats
- Digesting XML data loaded from a server
- Using open APIs in concert

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

"Mashing it up" doesn't mean we'll be heading into the kitchen to make a potato dish, nor does it mean we'll be heading out onto the lighted floor to participate in the latest dance craze. As applied to web applications, a *mashup* is a web page that combines content from multiple sources. While the term "mashup" may be fairly new, the concept is not. However, the advent of Ajax, along with the increasing availability of open APIs (see chapter 12), makes mashups easier than ever to create.

In this chapter we'll leverage the knowledge you gained about open APIs in chapter 12, as well as the Prototype JavaScript library that we examined in chapters 3 and 4 and have used throughout this book, to create a mashup of our own using the Yahoo! Maps and Flickr photo services open APIs.

If you haven't read chapters 3 and 12 yet, it might be a good idea to go back and do so before proceeding with this chapter. In particular, you should read or review the sections that discuss Prototype, Yahoo! Maps, and Flickr.

13.1 Introducing the Trip-o-matic application

Half the fun of taking a trip—sometimes even more than half—is bragging about it afterward. Often, this bragging is accompanied by a large collection of photos, which (depending on the skill of the person behind the camera) can be a spectacular showcase of photographic skill, or an abysmal collection of boring and off-focus images.

In either case, the power of the Internet now gives anyone the ability to force these photos onto a much larger audience than just family and friends who have no choice but to sit helplessly while you drag out the slide projector and screen.

13.1.1 Application purpose

One thing that can help keep trip photos from being an exercise in monotony is to give them *context*. And to help us achieve this context, we'll write a web application that could only be called the *Trip-o-matic*.

At this point you might be thinking, "Why not just put the photos up on a Flickr account and be done with it?"

Indeed, we'll actually be doing just that. But that's not quite good enough to achieve the context we crave. Sure, we could tell visitors where the photos were taken by adding photo comments like "Here are the photos from Austin," "Here are the photos from Oklahoma City," "Here are the photos from Topeka," "Here are the photos from Lincoln," and "Oh, look! It's Mount Rushmore!"

But we'll do one better than that: *we'll show maps*!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13.1.2 Application overview and requirements

The Trip-o-matic application, while quite simple, *is* involved enough to lay the foundations for mashup applications that are more complex. The application will consist of a single page whose content is entirely driven by a data file that contains the information for a particular trip that we have made. Since the *same* page will be used no matter which trip is to be shown, it is imperative that no trip-specific information be hard-coded onto the page or into the application code. *All* trip information will be garnered from the data file that the page will read.

This might remind you of server-side template applications using technologies such as JSP or PHP. But our application will consist entirely of client-side code with one exception: we'll need to employ the services of the cross-server proxy servlet that we developed in section 12.1.2. This proxy will allow us to escape the confines of the Ajax security sandbox and make cross-server requests to the Yahoo! and Flickr Photo Services sites.

The data file that we'll feed into this application contains all the information we'll need to know about a specific trip. A short name, a description, and even information regarding the access keys to our Yahoo! and Flickr accounts will be part of the data provided by this file.

The trip information will also include a list of *points of interest* along the path of the trip. For each such point, we'll provide a short name, a longer description, the location of the point, and information on how to retrieve the Flickr photos associated with that point of interest.

Upon page display, the application will read the trip information and show a list of the short names for the included points of interest. A visitor's click on an entry in this list will cause a map showing the area around the point to be displayed. Clicking on the map will display the thumbnails for the photos associated with the point, and clicking on a thumbnail will show the full-sized photo for that thumbnail.

There's nothing like making something interactive to help keep it interesting!

Unlike the shorter examples shown in chapter 12, we'll be applying a moderate amount of styling to the application page in order to achieve a rudimentary level of usability. But like those examples, this application would definitely benefit from the attentions of a professional UI expert, and could certainly use the help of a good visual designer.

Also, in interest of brevity (you'll be thankful), we're going to be giving short shrift to error checking. If this example at all intrigues you, your first enhancement to it should be to beef up the practically nonexistent error handling.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

As it stands, this example will be enough to serve as a "proof of concept" and cover all of the technical ground we'll need to get such an application on the road (pun intended)!

13.2 The Trip-o-matic data file

Before we write a single line of code, we need to design the data file that we'll feed to the page. This file will reside on the server and will be retrieved via an Ajax request upon page load, at which time it will be digested and client-side JavaScript structures representing the contained information will be created.

The specific file to be loaded will be identified via a query parameter that the application will expect to have been passed on the URL.

13.2.1 What format should we use?

As you've seen in previous chapters, responses to Ajax requests can take the form of plain text, HTML fragments, JSON notation, or an XML document. Which should we choose as the best format for our trip data?

Since the trip data is obviously a set of structured information, plain text and HTML are dismissed out of hand as the former has no means to represent the structure of the data and the latter is a display, not a data, format. This leaves us with the choice of JSON or XML.

JSON is an attractive notation because it is so easy to digest in the client program; a simple call to the `eval()` function transforms the JSON response into the corresponding JavaScript structures. JSON is also easy for program code to generate. But JSON isn't necessarily the best choice for larger data sets that will be hand-coded, as we know our trip data files will be.

JSON notation is terse and uses simple delimiter characters such as square brackets and curly braces to indicate arrays and structures. JSON data that consists of nested information can quickly become confusing and start to look like a data stream full of line noise (or that your pet iguana walked across your keyboard). Not only can this be visually difficult to parse, but it also becomes easier to introduce unintended errors into the data during revisions or additions.

XML, on the other hand, requires a bit more client-side work to digest in JavaScript code, but it *is* better-suited to hand-coding as its wordier markup nature makes it easier for people to mentally inspect the structure of data within the document. This makes the data easier to create, maintain, and revise.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

So, for choices, we are faced with

- JSON is easy to digest in code, but not so easy to hand-build.
- XML is more complicated to digest, but easier to hand-build.

Since we are going to write the digesting code *once* but will potentially be writing *many* trip data files, we'll make things easy on ourselves and choose XML since it simplifies the task that we'll be performing more often.

That's not being lazy. That's just being smart!

Besides, we're going to set things up so that, if we have chosen unwisely, it will be easy to back out of this decision.

13.2.2 The trip data format

A trip XML document will contain the information for a single trip. The scalar data that we'll need for each such trip consists of the following:

- A title for the trip
- The Flickr API key (see section 12.3)
- The Flickr NSID for the account holding the trip pictures (also see section 12.3.1)
- A text string to use as a header for the points of interest
- A description of the trip

The root element of our XML document format will be a `<trip>` element, and the first four scalar data elements will be specified as the attributes of that element. Since the description data is likely to be rather long and contain special characters, and perhaps even HTML markup, we'll provide it within its own child `<description>` element. The text body of this `<description>` element can then be contained within a CDATA section, when necessary, to allow embedded HTML markup, which would otherwise cause XML parsing issues.

A typical `<trip>` element and its `<description>` element might look like this:

```
<trip title="The Trip Title"
      flickrNSID="97545223@N00"
      flickrKey="78eca5287f3f05397dfba77ef40c7df53"
      poiTitle="Where we went">
  <description>
    <![CDATA[
      This is a descriptive comment for the trip complete with
      <b>some HTML markup</b>.
    ]]>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
 </description>
    ... <!-- other child nodes go here -->
</trip>
```

In addition to the <description> element, the <trip> element contains a list of child elements, each of which defines one of the trip's points of interest. The information provided for each point of interest (in a <poi> element) will be

- A short name for the point (used as the text for the on-page list of points)
- The latitude of the point location
- The longitude of the point location
- The Flickr photo set ID for the photos associated with this point
- A description or comment for this point

All but the description information for a point of interest will be provided by attributes to the <poi> element. Because the description information for a point of interest can also contain HTML markup, the description of the point will be specified by a text or CDATA section in the body of the <poi> element.

A typical <poi> element could then be

```
<poi name="Austin, TX"
     latitude="30.266748"
     longitude="-97.74176"
     photoSetId="1151001">
  <![CDATA[
    We started in Austin, TX on May 23, 2006. It was a
    <i>gorgeous</i> and sunny day. Little did we know what
    was in store for us...
  ]]>
</poi>
```

For those who didn't use a GPS to record the latitude and longitude of each point on the trip, conversion from a location string to coordinate values can be achieved using the Yahoo! Geocoding API that we examined in section 12.1.3.

Next, let's see how to set up our Flickr photos so that the appropriate value for the photoSetId attribute can be easily accessed by the Trip-o-matic application.

13.2.3 Setting up Flickr photo sets

One of the reasons that the Flickr photo service has grown so incredibly popular in a short period of time is that it's just so dang easy to use. Signing up for an account is simple and quick (if you already have a Yahoo! account, you can even use that), and you can start uploading photos right away using their web interface. If you

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

want to make that process even easier, you can download their desktop application upload tool (that allows you to easily perform batch uploads), which is available for Windows as well as Mac OS X. There are plug-ins for iPhoto or Windows Explorer that make it even easier to upload your photos. There's even a means of uploading your photos via email. It doesn't get much easier than that.

In the example we worked through in section 12.3 we obtained the information for *all* the public photos uploaded to the target account. That's clearly not what we want to do here. Rather than unleashing a torrent of photographic imagery, we want to limit ourselves to only a subset of the uploaded photos: those we took at one of our trip's points of interest.

One way that Flickr allows us to organize our photos, and that we'll take advantage of, is the ability to create *photo sets* and add any of our uploaded photos to that set. Since adding a photo to a set does not impact any other uses of the photo, or its ability to be added to other sets, we can use the set capability to our advantage without fear of adding artificial constraints on the use of our Flickr photos for other purposes.

Creating and managing photo sets is as easy as everything else with Flickr. Simply click the Organize tab and take it from there, using Flickr's Organizer interface to create and manage your photos sets. Once your trip photos are uploaded, create a photo set for each point of interest in the trip data file, and then add the appropriate photos to that set.

In our application, the Flickr method that we'll be using to obtain the photo set information requires that we reference the set by its *ID* rather than by name. This ID is not exposed on the Flickr website (at least, we couldn't find a means of displaying it), but the Flickr API provides a method, `flickr.photosets.getList`, that allows us to obtain the information for all our groups, including their ID values.

Since this isn't something we'll be looking up every time, it's not necessary to write any code to obtain this information; just hit the Flickr service with a URL like the following:

```
http://www.flickr.com/services/rest/?method=flickr.photosets.getList
&api_key=78eaa5287f3f0b37dfb47def40c7df13&user_id=95355920@N00
```

and inspect the results for the ID of the photo sets to be referenced. Of course, in this URL, be sure to substitute your *own* values for the API key and NSID of the Flickr account to be targeted.

With that, it's time to start writing some application code!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13.3 The TripomaticDigester class

We have the Prototype toolkit at our disposal, we've seen lots of examples of advanced JavaScript throughout this book, we've set up our Flickr photo sets, and we've decided on the format for our trip data file. So let's get down to business!

One of the first tasks that we're faced with is digesting the trip data file that we defined in the previous section. Rather than just including all the script necessary to obtain the XML document, digest it, and create the JavaScript structures that will hold the trip data directly on the page, we'll create a JavaScript object to handle that task.

In fact, we're going to keep this task completely separate from the rest of the Trip-o-matic application. But why?

Keeping the data digestion code separate from the application logic will not only help keep the code organized, but abstracting this duty completely from the rest of the application code *decouples* the task of reading the data from the task of processing the data, and will allow us to revisit the data file format at any time in the future. Should we ever want to change the XML format, or even decide that XML was a poor choice and that we should have gone the JSON route, only the digester class will be affected. This level of abstraction, a teeny-tiny example of the concept of *separation of concerns*, is *de rigueur* for code written in server-side languages, and our client-side JavaScript deserves no less respect.

We'll call our digester class *TripomaticDigester*, and it will be defined in the `TripomaticDigester.js` file. Next, let's examine each part of the implementation of that class.

13.3.1 The dependency check

Have you ever gotten some really cryptic error message from the JavaScript engine in your browser only to eventually discover that it was caused by a simple missing dependency? We've all wasted hours scratching our heads, trying to figure out what's wrong with our JavaScript code, and then kicked ourselves for forgetting to include some `.js` file that our code depended on.

Our digester, as well as the rest of the JavaScript code for this application, will rely heavily on the Prototype library. Rather than letting the browser issue its typically cryptic error message in the event that the Prototype library is not loaded prior to importing our digester class, we can perform an explicit check and issue our own, hopefully clearer, message.

So at the head of our digester script file, we place the following:

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
if (!Prototype) {
  throw new Error(
    "Prototype must be in scope to use Trip-O-Matic");
}
```

The Prototype library defines an object named Prototype, and we check for its existence. If it is not defined, we issue a JavaScript error that will appear on the JavaScript console (or in a pop-up window, depending on the browser used), telling us clearly that we've been remiss.

If we want to be really picky, the Prototype object contains a property that declares the version level of the Prototype library, and we could check that property as well.

This is a handy technique that can be used anywhere that dependencies on other JavaScript libraries need to be checked.

13.3.2 The TripomaticDigester constructor

The job of the TripomaticDigester constructor is to take the URL of a trip data resource, digest the XML document identified by that resource, store the digested trip data for later retrieval, and inform the caller when the process is complete. This is no small feat, but by taking things one at a time and delegating some tasks to implementation functions, we'll see that it's not as bad as it may sound.

Because we are using the Prototype library to create our JavaScript class, the constructor for the TripomaticDigester consists of creating the class and defining an initializer in the class's `prototype` property, as shown in listing 13.1. Note that this listing is not the complete code for the class. We'll be looking at the class one piece at a time.

Listing 13.1 The TripomaticDigester constructor components

```
TripomaticDigester = Class.create();

TripomaticDigester.prototype = {

  initialize: function(dataUrl,onLoadHandler) {      ❶ Stores initialization
    this.dataUrl = dataUrl;                             parameters
    this.onLoadHandler = onLoadHandler;
    new Ajax.Request(           ❷ Fetches data file
      this.dataUrl,               contents via Ajax
      {                                              ❸ Binds onSuccess
        onSuccess: this.onDigest.bind(this),           handler to instance
        onFailure: function() {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        throw new Error('failed to load ' + this.dataUrl);
      }
    }
  );
},
```

The code doesn't look too bad, but truth be told, it's not doing a whole lot of the hard work either. The `initialize()` method accepts the URL of the trip data resource and a function reference to serve as a handler to be notified when the trip data has finished loading. After storing these values ❶, the constructor fires off an Ajax request to fetch the contents of the specified URL ❷.

Note that the `onSuccess` event handler for this request ❸ is specified using the Prototype `bind()` method to make sure that the context object for the handler method `onDigest()` is the current instance of the TripomaticDigester.

Refer back to section 3.1.2 if you need to review what context object binding is all about and what it has to do with iguanas.

The real work starts when the `onDigest()` callback method is invoked.

13.3.3 Digesting the trip data

If the URL supplied to the constructor was valid, and nothing else goes awry, the `onDigest()` method will be called when the XML document has been received from the server. The implementation of this method is shown in listing 13.2.

Listing 13.2 The `TripomaticDigester.onDigest()` method

```
onDigest: function(request) {
  var xmlDoc = request.responseXML;
  var tripElement = xmlDoc.childNodes.item(0);
  if (tripElement.nodeName != 'trip')
    throw new Error( 'root element must be <trip>' );
  this.title = tripElement.getAttribute('title');
  this.flickrNSID = tripElement.getAttribute('flickrNSID');
  this.flickrKey = tripElement.getAttribute('flickrKey');
  this.poiTitle = tripElement.getAttribute('poiTitle');
  this.description = '';
  var self = this;
  var descriptionElements =
    tripElement.getElementsByTagName('description');
  $A(descriptionElements).each(function(descriptionNode) {
   self.description += self.collectText(descriptionNode);
  });
  this.points = new Array();
  var poiElements = tripElement.getElementsByTagName('poi');
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    $A(poiElements).each(this.loadPoint.bind(this));
    this.onLoadHandler(this);
  },
```

The task performed by this function is important, but it's a straightforward matter of digesting XML. The XML document is obtained from the `request` parameter (an instance of XMLHttpRequest). Its first (and only) child is obtained, and a check is made to verify that this root element is a `<trip>` node as expected.

If the check succeeds, the attributes of the element are obtained and stored in instance variables of the digester object. Then the text content of the `<description>` child elements is collected and stored by invoking the `collectText()` implementation method (which we'll describe soon). Note that we use the Prototype `$A()` function to convert the NodeList to an Array, and then invoke the `each()` method on that Array in order to collect the content of all `<description>` nodes.

Wait a minute! *Nodes*? More than one?

Yes. We're going to be lenient here and allow the user to spread the description of the trip over multiple elements if desired. We could be mean and restrictive, allowing only a single `<description>` element—but why not just be nice?

The `<poi>` child elements are then collected, again with the assist of the Prototype `$A()` and `each()` functions to invoke the `loadPoint()` method on each `<poi>` element (after binding the reference to the digester object instance). This is a good example of how the `each()` method of the Prototype Enumerable class helps us keep code manageable by allowing us to easily separate out the complex processing for the array elements into another function.

That method, `loadPoint()`, is tasked with digesting the `<poi>` elements. You'll learn more about that method in the next section.

Finally, the client handler registered with the constructor is invoked, passing the instance of the digester to this function. This eliminates any need for that handler to hook up to the digester via a global variable or other external means.

Now let's move on to the really interesting data: the aptly named points of interest!

13.3.4 Loading the points of interest

In the `onDigest()` method in listing 13.2, we called an iterator method named `loadPoint()`for each `<poi>` element found nested in the `<trip>` element. That method, bound to the instance of the digester, has the job of creating a JavaScript object describing each point of interest by gathering the information in the passed element. Its implementation is shown in listing 13.3.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Listing 13.3 The TripomaticDigester.loadPoint() method

```
loadPoint: function(poiElement,index) {
    this.points.push({
      name: poiElement.getAttribute('name'),
      latitude: poiElement.getAttribute('latitude'),
      longitude: poiElement.getAttribute('longitude'),
      photoSetId: poiElement.getAttribute('photoSetId'),
      description: this.collectText(poiElement)
    });
  },
```

Creating a JavaScript object from the attributes of the passed element is almost trivial; the `getAttribute()` DOM method is called on the element for each value, and a correspondingly named property is created in a JavaScript object. The newly created object instance is then *pushed* to the end of the `points` array instance variable for later reference.

As with the description data for the `<trip>` element's `<description>` child, the `<poi>` element's description data is collected from the element via the `collectText()` implementation method.

What's all *that* fuss about?

13.3.5 *Collecting element text*

Determining the value of an attribute of an element in an XML document is almost trivial, as we have seen in the methods that we've already created in our digester class. Only slightly more involved is obtaining the child elements of any particular element. But gathering text data from the body of an element poses just a bit more of a challenge.

Unlike attributes or child elements, there is no *one* DOM method that gives you all of the text that lies within the body of an element. And if you think about it, it's easy to understand why. Consider the following XML document fragment that might exist in our document:

```
<description>
  The quick young cub jumped over the lazy bear.
</description>
```

This seems fairly straightforward, and most XML parsers would return a single text node for the `<description>` element's body content. But what about something like the following?

```
<description>
  The quick young cub <!-- was he that quick? --> jumped
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    over the lazy bear.
  </description>
```

In the this fragment a comment node has been added, which causes the XML parser to split the text into (at least) two text nodes separated by a comment node.

Or what about the following?

```
<description>
  <![CDATA[
   The <strong>quick</strong> young cub jumped over the
  <i>lazy</i> bear.
  ]]>
</description>
```

In this fragment the body text has been entered as a CDATA section so that it can contain characters that would otherwise introduce syntax errors into the document—in this case, HTML markup. CDATA information is delivered in its own node type.

How these issues are dealt with depends on the needs of the application. For our example, we'll collect all text and CDATA elements (ignoring comments) and concatenate them into a single text block to serve as the content of the node. This will apply to the description of the trip and the point of interest elements.

To this end, we create the `collectText()` method, whose implementation is shown in listing 13.4.

Listing 13.4 The `TripomaticDigester.collectText()` method

```
collectText: function(element) {
  var text = '';
  $A(element.childNodes).each(
    function(child) {
      if ((child.nodeName == '#text') ||
          (child.nodeName == '#cdata-section')) {
        text += child.data;
      }
    }
  );
  return text.strip();
}
```

Given an element passed as the sole parameter, this method iterates over all children of the element looking for text and CDATA nodes, identified with node names of `#text` and `#cdata-section`, respectively. Any such nodes that we find have their content concatenated into a text variable whose trimmed value

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

(trimming provided by the Prototype `String.strip()` method) is returned as the value of the method.

This completes our standalone digester class. Again, defining this process in its own class decouples the reading of the data from the rest of the application. This gives us the freedom to change the details of the digestion process to include the data format, without fear of introducing errors into the application.

13.4 The Tripomatic application class

Now that we've got digesting the data out of the way, let's turn our attention to the application itself. We could just code a page, with embedded JavaScript, to be our application. But we're smarter than that. Instead, we're going to set up a JavaScript class that implements the application in keeping with the object-oriented theme that we've endorsed throughout this book.

This application has a lot to do and a lot to keep track of. There's going to be trip information, points of interest, maps, as well as thumbnails and photos to create, manage, and manipulate. But by taking things one step at a time, and using the power that object-oriented programming, JavaScript, and the Prototype library gives us, we'll get there.

The list of things this application needs to do includes

- Create the DOM elements in which the application content will be displayed. This includes elements for
  - The trip title
  - The trip description
  - The points of interest, each of which will be a clickable element
  - A header to label the point of interest list
  - The map showing a clicked point of interest
  - A set of thumbnail images for the photos taken at a point of interest
  - A full-sized photo of a clicked thumbnail
- Assign event handlers so that
  - A click on a point of interest displays the corresponding map using Yahoo! Maps
  - A click on a displayed map fetches and displays the thumbnails images in the photo set associated with the map's point of interest using Flickr services
  - A click on any thumbnail image displays the full-sized photo for the thumbnail, again using the Flickr service

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

And all of this needs to be accomplished in an object-oriented fashion without the use of global variables or any hard-coded element IDs. Let's start by setting up the Tripomatic class and its constructor.

13.4.1 The Tripomatic class and constructor

To start, we create a file named `Tripomatic.js` and populate it with the same type of dependency checks that we were using in the TripomaticDigester class. Again, using the Prototype style of creating classes, we set up the skeleton of the class as shown in listing 13.5. This listing is far from the complete definition of the class, but we'll be adding to it throughout this section until we have the complete and working application. Subsequent listings in this section will generally not repeat code that is already shown; they will depict newly added code.

Listing 13.5 Skeleton for the Tripomatic class

```
if (!Prototype) {
  throw new Error(
    "Prototype must be in scope to use Trip-O-Matic");
}

if (!TripomaticDigester) {
  throw new Error(
    "TripomaticDigester must be in scope to use Trip-O-Matic");
}

Tripomatic = Class.create();

Tripomatic.prototype = {
  /* instance methods will go here */
}
```

As is usual with Prototype-defined classes, the actual construction code will be placed in a method named `initialize()`. The parameters to this method will be the URL for the data file containing the trip information, the DOM element in which to create the application content, and an options hash for passing any optional information. This method is shown in listing 13.6.

Listing 13.6 The `Tripomatic.initialize()` method

```
initialize: function(dataUrl,container,options) {
  this.container = $(container);
  this.options = Object.extend(
    {
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
        enablePanAndZoom: false
      }, options
    );
    this.createContent();
    this.digester = new TripomaticDigester(
      dataUrl,
      this.onDataLoaded.bind(this));
    this.map = new YMap(this.mapContainerElement);
    if (this.options.enablePanAndZoom) {
      this.map.addPanControl();
      this.map.addZoomLong();
    }
  },
```

This method is deceptively simple because it delegates all the hard work to another method. The passed `container` is recorded in an instance variable, and the passed `options` are merged with the default options. Note that this class defines but a single option, `enablePanAndZoom`, which specifies whether the pan and zoom controls should be added to the Yahoo! Maps map.

But if we only have a single option, why bother with an object hash? Why not just define an optional third parameter? The answer: extensibility. With an application of this complexity, the chances that we'll eventually want to add more options are good. And if we were to not use an options hash at this point, when we changed the parameter to a hash later in order to accommodate extra options, the signature of the constructor would change, thus breaking the code of everyone who has used the class in their pages prior to the addition. By anticipating this need, we can add future options without changes to the constructor signature, thereby delighting the users of our class by avoiding unnecessary changes.

The method then calls an implementation method named `createContent()`. This method will create all the elements needed to display the data content of our application. We'll be inspecting it next.

After the content elements are created, an instance of TripomaticDigester is created using the `dataUrl` that was passed, and this instance is tucked away for later reference. A method named `onDataLoaded()` will be invoked when the digester has finished its job. Note how we have bound the function context (`this`) to this callback so that this instance of Tripomatic will also be the function context when `onDataLoaded()` is invoked,

Finally, the Yahoo! Maps map is created and initialized in a container that, presumably, was created by `createContent()` (which is indeed the case as we'll see shortly).

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

At this point, the application is completely initialized and ready for user interaction. But a lot went on behind the scenes in the `createContent()` and `onDataLoaded()` methods. Let's take a look at just what those methods did for us.

13.4.2 Creating the content elements

Now we come to the "*create or attach*?" conundrum. We could deal with the content of our application in two ways. One way, which we've seen used by the Button class example of chapter 3, is to have the user define the HTML elements, which our class would *attach* to as part of its initialization.

This tactic works well when the HTML either is very simple or has too many variations for us to guess at creating it ourselves. But in cases where a fairly complex, but predictable, element hierarchy is needed, it's sometimes better to have the script *create* the elements dynamically.

This latter strategy is the way that we'll define our Trip-o-matic application. And it does it via the `createContent()` method that we called from the constructor.

This method is responsible for creating a DOM hierarchy, nested inside the container element that the user passes to us, equivalent to the HTML snippet shown here:

```
<h1></h1>
<h2></h2>
<div class="tripomaticPoiContainer">
  <h3></h3>
  <ul></ul>
</div>
<div class="tripomaticMapContainer"></div>
<div class="tripomaticPoiDescription"></div>
<div class="tripomaticThumbsContainer"></div>
<div class="tripomaticPhotoContainer"></div>
```

It is into these elements that the data content of our trip will be inserted.

The `<h1>` and `<h2>` elements will be used for the trip title and description, respectively. The `<div>` element containing the `<h3>` and empty `<ul>` elements will hold the list of points of interest. The string supplied by the `poiTitle` attribute of the `<trip>` element will be placed in the `<h3>` element, and the `<ul>` element will be populated with the point-of-interest items.

Each `<div>` element is assigned a unique CSS class name (all of which are prefixed with the string *tripomatic* so as not to pollute the user's CSS class namespace) that not only helps identify the use to which the element will be put, but also allows the user to style every element created. This is an important point when creating elements on behalf of the user. We must be sure that the user is able to

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

address each element via a CSS selector. Any element that is not selectable cannot be styled by the user. The non-<div> elements do not need the class name treatment as they are unique within the construct and can be addressed using element name CSS selectors.

As it's usually a good idea to avoid hard-coding strings in code, we want to factor the strings that represent the class names out of the code and into class-level references. As we discussed in chapter 3, we can emulate class-level "constants" by assigning them directly as properties of the class. Of course, we know that they're not really constants, but we'll treat them as if they were. These assignments are placed in the `Tripomatic.js` file between the class and `prototype` declarations, as shown in listing 13.7.

Listing 13.7 The class name "constants"

```
Tripomatic = Class.create();

Tripomatic.CLASS_POI_CONTAINER = 'tripomaticPoiContainer';
Tripomatic.CLASS_MAP_CONTAINER = 'tripomaticMapContainer';
Tripomatic.CLASS_POI_DESCRIPTION = 'tripomaticPoiDescription';
Tripomatic.CLASS_THUMBS_CONTAINER = 'tripomaticThumbsContainer';
Tripomatic.CLASS_PHOTO_CONTAINER = 'tripomaticPhotoContainer';

Tripomatic.prototype = {
```

With that, we're ready to examine the method (listing 13.8) that creates the DOM elements.

Listing 13.8 The `Tripomatic.createContent()` method

```
createContent: function() {
  this.container.innerHTML = '';
  this.tripTitleElement = document.createElement('h1');
  this.tripDescriptionElement = document.createElement('h2');
  this.poiContainerElement = document.createElement('div');
  this.poiTitleElement = document.createElement('h3');
  this.poiListElement = document.createElement('ul');
  this.mapContainerElement = document.createElement('div');
  this.poiDescriptionElement = document.createElement('div');
  this.thumbsContainerElement = document.createElement('div');
  this.photoContainerElement = document.createElement('div');
  this.container.appendChild(this.tripTitleElement);
  this.container.appendChild(this.tripDescriptionElement);
  this.container.appendChild(this.poiContainerElement);
  this.container.appendChild(this.mapContainerElement);
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    this.container.appendChild(this.poiDescriptionElement);
    this.container.appendChild(this.thumbsContainerElement);
    this.container.appendChild(this.photoContainerElement);
    this.poiContainerElement.appendChild(this.poiTitleElement);
    this.poiContainerElement.appendChild(this.poiListElement);
    this.poiContainerElement.className =
      Tripomatic.CLASS_POI_CONTAINER;
    this.mapContainerElement.className =
      Tripomatic.CLASS_MAP_CONTAINER;
    this.poiDescriptionElement.className =
      Tripomatic.CLASS_POI_DESCRIPTION;
    this.thumbsContainerElement.className =
      Tripomatic.CLASS_THUMBS_CONTAINER;
    this.photoContainerElement.className =
      Tripomatic.CLASS_PHOTO_CONTAINER;
  },
```

The `Tripomatic.createContent()` method may be a tad on the lengthy side, but it's fairly straightforward. First, the container passed into the constructor by the page author is cleared, and then the various elements that will be required by the application are created and stored in properties of the instance.

The method then hooks up the elements into the hierarchy that we depicted as HTML earlier in this section, as children of the page author's container. Finally, the CSS class names are assigned to the created elements.

Well, that wasn't too bad. But this just creates a set of empty elements devoid of any actual trip data. Let's see how we'll handle obtaining and filling in the data.

13.4.3 Filling in the trip data

If you recall, one of the steps we took in our `initialize()` method was to create an instance of a TripomaticDigester. We gave it the URL of a trip data file and a reference to a method named `onDataLoaded()` to call when it has digested the trip data file.

We created the digester instance *after* we created the content elements so that when `onDataLoaded()` is called, we know that it's safe to assume that the DOM hierarchy has already been assembled. Let's take a look at the code for this method (listing 13.9).

Listing 13.9 The `Tripomatic.onDataLoaded()` method

```
  onDataLoaded: function(digester) {
    this.tripTitleElement.innerHTML = digester.title;
    this.tripDescriptionElement.innerHTML = digester.description;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    this.poiTitleElement.innerHTML = digester.poiTitle;
    digester.points.each(this.makePointOfInterest.bind(this));
    this.showPoint(digester.points[0]);
  },
```

The digester helpfully passes a reference to itself to the `Tripomatic.onDataLoaded()` method, and we use that reference to obtain the digested data. But even if the digester had not been that helpful, we still would have been OK, as this method is bound to the current Tripomatic instance, giving us access to the digester reference stored as an instance property. Ah, the joys of object orientation!

Displaying the trip title and description is a simple matter of setting the `innerHTML` property value of the appropriate content elements (that we set up in the `createContent()` method) with their corresponding values. We then use the Prototype `each()` method on the list of points, specifying the `makePointOfInterest()` method as the iterator function.

Just before relinquishing control to the page visitor (remember, all this will be happening as a result of the page load), we call a method named `showPoint()` with a reference to the first entry in the points of interest list. This will cause the information for the first point in the list to be displayed, and is the same function that will be used to handle a visitor click on any point name. We'll be looking at its implementation in section 13.4.4.

For each point that the digester loaded, we caused the `makePointOfInterest()` method to be invoked as an iterator function. This method is tasked with creating an active entry in the list of points that a visitor can click on to load the map for that point. This method's implementation is shown in listing 13.10.

Listing 13.10 The `Tripomatic.makePointOfInterest()` method

```
  makePointOfInterest: function(point) {
    var pointItem = document.createElement('li');
    pointItem.appendChild(document.createTextNode(point.name));
    pointItem.onclick = this.onPoint.bindAsEventListener(this);
    pointItem.point = point;
    this.poiListElement.appendChild(pointItem);
  },
```

The `Tripomatic.makePointOfInterest()` method, using the DOM manipulation API, creates an `<li>` element to contain the point entry. An event handler for the `click` event is established as a method named `onPoint()`, bound as an event listener to the current instance. So that the point information associated with this

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

item can be easily obtained by the event handler, we tack it onto the `<li>` element as a property named `point`. The created `<li>` element is then appended to its parent, the `<ul>` element that we had created earlier during initialization.

The invocation of this function on the last point of interest completes the rather long chain of events that occurs when the page is loaded. When supplied with the sample XML data file found in the downloadable source code for this chapter at www.manning.com/crane2, the result is displayed to our visitor as shown in figure 13.1.

Note that the title, description, and points of interest have been displayed, and the map for the first point has been loaded. The empty areas at the bottom of

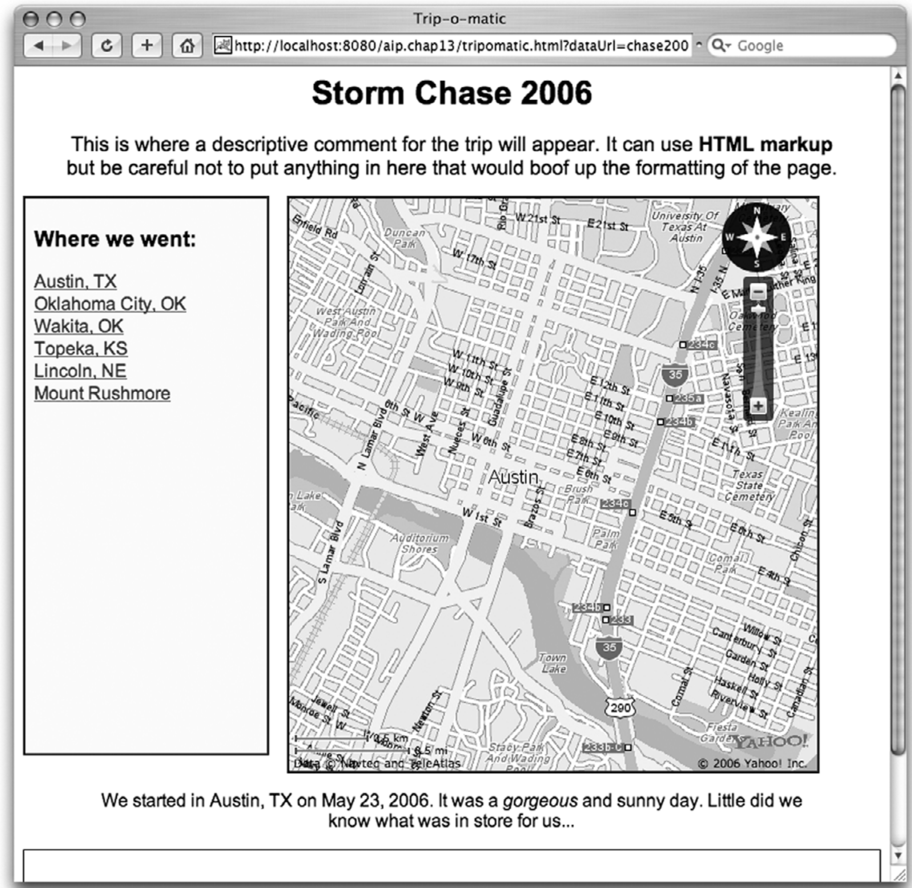

Figure 13.1 We're in Austin!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

the page are where the strip of thumbnails will be loaded, and where a thumbnail will be expanded to a full-sized photo upon a click.

But how did that map get there?

13.4.4 Showing the map

Because of the event handler that we set on the point-of-interest entries, a visitor click on one of the point names triggers the `onPoint()` method. As an event handler, this method is called with an Event instance as its parameter, from which we can determine which element was clicked on. As you'll recall, we set the point information on that element in a property named `point` for easy access by this method. The code for this event-handler method is shown in listing 13.11.

Listing 13.11 The `Tripomatic.onPoint()` event-handler method

```
onPoint: function(event) {
  this.showPoint(Event.element(event).point);
},
```

The only action that this handler performs is to locate the point information on the event's target element and pass it to the `showPoint()` method. The `showPoint()` method is what causes the map associated with a point of interest to be drawn. It is invoked whenever a user clicks on an entry in the points-of-interest list and, as you may recall, when the data is initially loaded. Let's take a look at its implementation in listing 13.12.

Listing 13.12 The `Tripomatic.showPoint()` method

```
showPoint: function(point) {
  this.currentPoint = point;
  var geoPoint = new YGeoPoint(point.latitude,
                               point.longitude);
  this.map.drawZoomAndCenter(geoPoint,4);
  this.mapContainerElement.onclick =
    this.showThumbnails.bindAsEventListener(this);
  this.mapContainerElement.point = point;
  this.poiDescriptionElement.innerHTML = point.description;
  this.thumbsContainerElement.innerHTML = '';
  this.photoContainerElement.innerHTML = '';
},
```

After recording the point passed to this method as the *current point*, our method creates a YGeoPoint instance and uses it to display the YMap centered

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

on the point-of-interest's location, just as we explored in the example in section 12.1.1.

We then establish a click event handler on the map element that binds a method named `showThumbnails()` as the event handler. We'll be looking at that method next.

The description of the point is displayed and the containers for the thumbnails and photo are cleared to remove any images left over from a previously displayed point.

Now we sit and patiently wait for a visitor to click on the map.

13.4.5 Loading the thumbnails

If a visitor ever figures out that clicking on the map does something interesting—remember, we already stated that this application wasn't a paragon of usability—they'll be rewarded for their intuitiveness. When the map is clicked, we want to fetch the information about the thumbnails stored in our Flickr account that we associated, via the photo set ID, with the point of interest whose map is currently displayed.

We've made Flickr requests before—check out section 12.3 for a refresher—so the code we'll write to accomplish this should look rather familiar even though we'll be using a different Flickr method when accessing the Flickr REST API.

Recall that we set up a Flickr photo set to correspond to each point of interest, and recorded the IDs of those sets in the trip data file. So rather than using the `flickr.people.getPublicPhotos` Flickr method to retrieve all public photos, we'll use the `flickr.photosets.getPhotos` method to limit the retrieval information to only the photos that we placed in the identified set. Figure 13.2 shows the bottom of the page after clicking on the map in order to show the associated thumbnails.

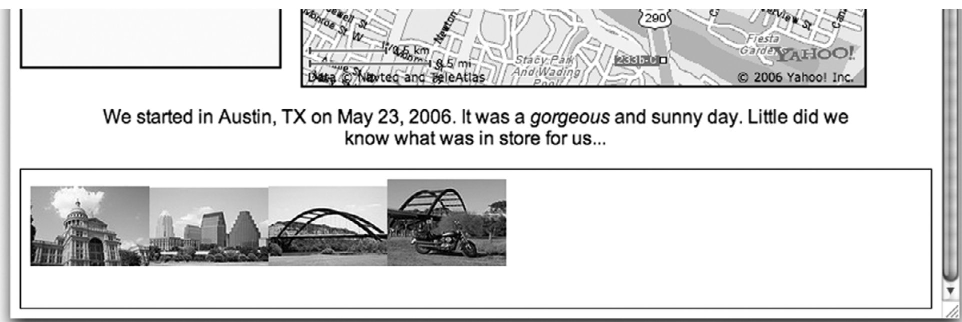

Figure 13.2 Thumbnails of Austin photos displayed

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

The event handler that performs this action in response to the click appears in listing 13.13.

Listing 13.13 The `Tripomatic.showThumbnails()` method

```
showThumbnails: function(event) {
    new Ajax.Request(
      '/aip.chap13/proxy',
      {
        onSuccess: this.onPhotosetList.bind(this),
        method: 'get',
        parameters: {
          '.serviceUrl.':
            'http://www.flickr.com/services/rest/',
          api_key: this.digester.flickrKey,
          method: 'flickr.photosets.getPhotos',
          photoset_id: this.currentPoint.photoSetId
        }
      }
    );
  },
```

In the same manner in which we've issued many other Ajax requests, this method contacts the Flickr service by way of our server-side proxy agent. Section 12.1.2 examines the need for, as well as the operation of, this proxy agent.

Conveniently, and quite deliberately, we stored the information for the currently loaded point of interest in an instance property named `currentPoint`. This handler can use that property to reference the `photoSetId` property of the current point without the need for global variable references or indexes into the point array.

The `onSuccess` event handler (listing 13.14) for the request, specified as the `onPhotosetList()` method, is charged with digesting the XML document that the Flickr service will send to us in response to the issued request. Let's take a look at its implementation.

Listing 13.14 Digesting the photo set member document

```
onPhotosetList: function(xhr) {
  var doc = xhr.responseXML;
  var status =
    doc.getElementsByTagName('rsp')[0].getAttribute('stat');
  if (status == 'ok') {
    this.thumbsContainerElement.innerHTML = '';
    var photos = doc.getElementsByTagName('photo');
    $A(photos).each(this.makeThumbnail.bind(this));
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    } else {
      throw new Error('getPhotos request failed');
    }
  },
```

Despite the fact that we called a different Flickr method than we used in section 12.3, the Flickr service returns an XML document of the same format. Digesting the document is therefore performed in the same manner.

First we obtain the response document, and check it to make sure the service reported a successful response. If all went well, we clear out any previous thumbnails, obtain the list of all `<photo>` elements in the document, and invoke the `makeThumbnail()` method as the iterator function upon each such element node. The `makeThumbnail()` method, shown in listing 13.15, is where all the heavy lifting to create the thumbnail images takes place. (Note that in the event of a failure we throw an Error instance constructed from the entire response text. In reality, we'd want to extract only the error message from the response document, but we don't do so here in the interest of brevity. As an exercise, how would you enhance this code with better error detection and recovery?)

Listing 13.15 The `Tripomatic.makeThumbnail()` method

```
makeThumbnail: function(photo,index) {
    var baseUrl = 'http://static.flickr.com/' +
                  photo.getAttribute('server') + '/' +
                  photo.getAttribute('id') + '_' +
                  photo.getAttribute('secret');
    var thumbUrl = baseUrl + '_t.jpg';                    ❶ Formats photo URL
    var photoUrl = baseUrl + '.jpg';
    var thumb = document.createElement('img');
    thumb.src = thumbUrl;
    thumb.style.cursor = 'pointer';
    thumb.onclick = this.showPhoto.bindAsEventListener(this);
    thumb.photoUrl = photoUrl;                              ❷ Stores URL for
    this.thumbsContainerElement.appendChild(thumb);            later reference
  },
```

As it turns out, all that heavy lifting isn't so complicated. Just as we did in section 12.3.2, we construct the URL to reference the thumbnail image hosted on the Flickr site, and then create an HTML `<img>` element that refers to that URL.

While we're generating the URL of the thumbnail image, we also generate the URL of the larger version of the photo ❶ that we'll want to show when the thumbnail is clicked, and store that as a property of the thumbnail `<img>` element ❷.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Later, in the `showPhoto()` method, which we set as the `onclick` event handler for the `<img>` element, we'll just use the recorded URL when creating the `<img>` element for the larger photo.

13.4.6 *Displaying the photos*

It seems like we've done a lot of work to get to this point, but finally we're ready to show pictures! The bottom of the page, with a fabulous photo displayed, appears in figure 13.3.

In our `makeThumbnail()` method of listing 13.15, we specified a method named `showPhoto()` as the `click` event handler for each thumbnail image element. This method (listing 13.16) is very similar to the corresponding function we set up in section 12.3.2.

Listing 13.16 The `Tripomatic.showPhoto()` method

```
showPhoto: function(event) {
  if (this.photoElement == null) {
    this.photoElement = document.createElement('img');
    this.photoContainerElement.appendChild(this.photoElement);
  }
  this.photoElement.src = Event.element(event).photoUrl;
}
```

Figure 13.3 The Austin skyline

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Recall that because this function is a `click` event handler for a thumbnail image element, that `<img>` element will be the event's target element when the function is invoked. Because we wisely preloaded a property on that element that specifies the URL of the photo to be shown as a result of clicking on the thumbnail, the work that this function needs to do is rather simple: it checks to see if an `<img>` element to show photos has already been created and, if not, creates one. After the element is either located or newly created, its `src` property is set to the photo to be displayed.

At this point you may be wondering why we didn't create the `<img>` element for the full-sized photo in the `createContext()` method? Why wait until this method?

At the time that the `createContext()` method was executing, we had no photo URL to display. In fact, we don't get such a URL until the page visitor clicks on a map and then clicks on a thumbnail created as a result of that click. If we were to create an `<img>` element without a `src` attribute, or with an empty one, some browsers would display a "broken image" icon until a valid `src` attribute value would be assigned. That's something we don't want displayed on the page.

And with that, our application class has finally been completed. Now to put it on an HTML page.

13.5 The Trip-o-matic application page

By this time you're probably pleading, "Can we *please* start writing the page?" Your pleas will be heard. It seems we've done a lot of work without actually writing any HTML for the application page itself. But all that hard work will pay off, and you'll see that, having already dispensed with all the necessary support code, all we have to do on the page is write the HTML, CSS, and JavaScript code that is focused on creating an instance of the application class.

13.5.1 The Trip-o-matic HTML document

Let's set up the HTML page for the application. The complete code for the page is shown in listing 13.17.

Listing 13.17 The trip-o-matic.html page

```
<html>
  <head>                                                  ❶ Imports styles,
    <title>Trip-o-matic</title>                             libraries, and classes
    <link rel="stylesheet" href="styles.css"/>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
    <script type="text/javascript" src=
      "http://api.maps.yahoo.com/ajaxymap?v=2.0&appid=YahooDemo">
    </script>
    <script type="text/javascript" src="TripomaticDigester.js">
    </script>
    <script type="text/javascript" src="Tripomatic.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        var dataUrl = document.URL.toQueryParams().dataUrl;
        if (dataUrl == null)
          throw new Error('the dataUrl parameter must be set');
        new Tripomatic(          <-- 2 Creates instance of application
          dataUrl,
          'tripomatic',
          {
            enablePanAndZoom: true
          });
      };
    </script>
  </head>

  <body>
    <div id="tripomatic"></div>     <-- 3 Declares container for application elements
  </body>

</html>
```

The <head> element ❶ is filled in with an import for a CSS style sheet that we'll use to apply rudimentary styling to the page, as well as imports for the various external JavaScript files that we need: the Prototype library, the Yahoo! Maps library, the digester class, and the Tripomatic class itself.

We also include a script element in which we define the `onload` event handler for the page. In this handler function, we obtain the data URL passed to us as the `dataUrl` query parameter, with the help of the Prototype `toQueryParams()` String method. If no such parameter is found, we complain.

Then an instance of the Tripomatic class is created ❷, specifying the data URL, a reference to the container in which the application elements are to be created ❸, and an object hash of the options we desire.

In the <body> of the page, we create an empty <div> element in which the application will create its elements.

That's it. Everything else is handled by the code that we've already written. How easy is that for the page author?

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

At long last, our application page has been completed! But before we head out to celebrate, the next section briefly discusses the content of the `styles.css` file that we linked in the header of our page. Take a gander at it if you are interested in seeing why the page laid out as it did.

13.5.2 Tripping along with style

The `styles.css` file that we use to style our page isn't remarkable by any stretch of the imagination. Its purpose is merely to bring the page into the realm of at least marginal usability while not complicating our discussion of the page's core functionality.

It was not the intent of this example to focus on the elements of good style or usability, but to discuss the technical mechanics of creating a mashup application. Feel free to use the `styles.css` file, as well as the HTML of the application page, as a springboard to a significantly better styled and more usable application.

One important thing that this style sheet does point out is that we adequately designed the DOM elements that we created for selectability. The page author has a lot of control over how the elements appear and lay out on the page as a result.

That said, the `styles.css` file is shown in listing 13.18.

Listing 13.18 The rudimentary style sheet

```
body {
  font-family: Arial,Helvetica,sans-serif;
  padding: 8px;
  margin: 0px;
}

#tripomatic h1,h2 {
  text-align: center;
}

#tripomatic h1 {
  font-size: 1.8em;
}

#tripomatic h2 {
  font-size: 1.1em;
  font-weight: normal;
  padding: 0px 32px;
}

.tripomaticPoiContainer {
  float: left;
  border: 2px ridge maroon;
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir


```
  background-color: #ffffcc;
  padding: 8px;
  width: 200px;
  height: 480px;
  overflow: auto;
}

.tripomaticPoiContainer li {
  cursor: pointer;
  list-style-type: none;
  margin-bottom: 2px;
  color: maroon;
  text-decoration: underline;
}

.tripomaticPoiContainer ul {
  margin: 0px;
  padding: 0px;
}

.tripomaticMapContainer {
  float: left;
  width: 440px;
  height: 480px;
  border: 2px ridge maroon;
  padding: 8px;
  margin-left: 16px;
  cursor: pointer;
}

.tripomaticPoiDescription {
  clear: both;
  margin: 0px 32px;
  text-align: center;
  padding: 16px 32px;
}

.tripomaticThumbsContainer {
  clear: both;
  border: 1px solid black;
  padding: 8px;
  height: 100px;
  overflow: auto;
  left: 8px;
  right: 8px;
}
```

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

13.6 Summary

In this chapter you saw that a *mashup* is not the result of a multicar collision, but a web page or application that combines content gathered from multiple sources. We worked through a small example, combining content from Yahoo! Maps and Flickr photo services, that demonstrated how the power of Ajax, when combined with open APIs, makes such mashup applications fairly easy to create.

We also created an application that was a self-contained JavaScript class that requires no code on any HTML page that plans to use it (beyond that necessary to instantiate the class, of course). So did we create an application or a *component*? In fact, since we cleverly used no global variables or element IDs in our application/component, it would even be possible to include more than one instance of it on a page. Although that might not make much sense with this particular component, you can apply the techniques used in this chapter to your own components that can be used multiple times per page.

This example brought together a lot of the techniques that have been discussed throughout this book. We relied heavily on object-oriented techniques, leveraged the use of Prototype, made many an Ajax request, used available open APIs via our cross-server proxy, and handled lots of events.

In addition, you saw how the various classes and functions in the Prototype library can help you write JavaScript that is more concise, modular, and well organized. An example that saw heavy use in our Trip-o-matic application was the `each()` method added by Prototype as an extension to JavaScript arrays. Moving the array element processing out of the typical `for` loop and modularizing it to iterator functions that accept the individual array elements as their parameter not only isolates the element processing code into a concise, easy-to-understand function, but it also promotes reuse—the iterator function can be called, passing any object of the appropriate type, from code other than loops.

We again used the Yahoo! Maps API to display our maps. There's a lot more to the Yahoo! Map API, so there's still more that you can do to make your applications that employ maps even more interactive.

You learned more about Flickr photo services, particularly in the area of organizing groups of photos using *photo sets*. Again, our discussion only touches the high points of the available functionality. Sites or applications wanting to use interactive photos can do much more with Flickr than we've explored here.

So look around at all the content that's available to you on the Web via open APIs. You're sure to find a combination that strikes a chord, prompting you to create a mashup of your own.

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Late-breaking news!

Recently a new Yahoo! Service named Yahoo! Pipes (http://pipes.yahoo.com) has become available and is the quintessential (at least for now) example of what mashups can bring to the party! It comes with a web-based graphical mashup builder with drag-and-drop that features programming primitives, default data sources (Google Base, Yahoo!, and so on), and much more. Check it out!

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

index

Symbols

$() 98
  in jQuery 243
  in Prototype 261
$A() 100, 476
$.ajax() 21
$.each(), in jQuery 246
$F() 99, 127
$H() 102
<abbr> 449
<script> tags 24

A

abort() 14
absolute positioning 377
abstraction 131
Accordion 260
  drawbacks 263
Accordion control 259
ActiveX control 4
adapter pattern 137
addEventListener 169, 174
Ajax 4
  business perspective 6
  canceling 354
  deeper issues 4
  delivering feedback 338
  dialects of 27
  disruptive technology 4, 235
  DOM limitations and 50
  form submission 203
  history of 4
  invisible 330
  origin of term 4
  progress, showing 345
  requests via Dojo 119
  requests via jQuery 136
  requests via Prototype 125
  REST, using 422
  security sandbox 421, 468
  timing out 351
  undo/redo and 300
Ajax Push 326
Ajax wrapper objects 21
Ajax-based
  application 8
  office suites 6
Ajax-enhanced legacy applications 8
Ajax.PeriodicalUpdater 134
Ajax.Request 18, 20, 27, 127
  carrying extra data 358
  property of Ajax.Requeston-Complete callback 20
  simplifying Ajax 21
AJAXSLT library
  limitations of 55
Ajax.Updater 131, 361
  Prototype library 24
AMASS 392, 409, 411–413
  persistance 409
  store and restore 411
Amazon services 464
anchor tags and JavaScript 243
animated icon 344
animation 318
  cautions 345
anonymous
  function 69
  object 84
Apache Axis
  .jws files 57
  SOAP and 57
Apache Jakarta Project 425
  HttpClient 425
Apache XMLBeans 55
application design
  data formats 469
  data independence 468
  decoupling 473
  delegation 474
  designing for extensibility 481
  self-containment 496
  Separation of Concerns 473
application marketplace 6
application state and back button 276
application workflow logic 34
application, customer display 393
application-initiated update 326
architectural tiers, impact of Ajax 8
Array.partition() 269
arrays 100
  associative 102
  Prototype extensions 100
asynchronous 70
attachEvent 170
Austin, Texas 419, 491
AxisServlet 57

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

B

back button 272
  enabling using Ajax 289
backspace key
  disabling 275
  text elements and 275
bandwidth 401
bar-code scanning 140
Basic Event Model 165
bind() 104
binding expression 330
blogs 443
bookmarking and Ajax 272
bookmarks
  hashes 281
  logical 280
Boolean expressions 87
Brad Neuberg 283
breadcrumb trails 237
brittleness 30
browser
  "throbber" 338
  default action 177
  detection 129
  differences 169
  history and Ajax 272
  navigation features, working with 280
  visual cues 338
built-in objects, extending 80
business logic 33–34
business rules 211
button 82, 132
  back 280
  disabling multiple clicks 356
  enabling and disabling 92
  forward 280
  handlers 72
  instrumenting 82

C

cache 388
  data 389
  fetching from 401
  security 414
call() 103
callback
  event handler 318
  functions 70, 103
  handler function 12
  mechanism and RSH library 295
cancelBubble 177
Cascading Style Sheets (CSS) 5, 344, 363, 387, 482, 492, 494
  namespace pollution 482
  run-time modifications 91
  shortfalls in Internet Explorer 244
Castor 55
categorization 235
  breadcrumb trails 237
  desktop applications 238
  navigation bars 237
  scaling 236
  sidebars 237
CD object representation 67, 109
CDATA 470, 478
cinematic effects 313
class constant 80
class methods 81
classification 235
  and categorization 236
  versus categorization 236
class-level
  declarations 79
  members 91
click 181
ClientManager 402
client-side
  maintenance of application history 288
  state management 283
  undo stack 306
  validation security 208
closure 74, 89, 349
code generation 30
collaboration 326
combining HTML and JavaScript 263
Comet 326
Command pattern 339
community-based application 337
Complete 13
complex structured data 34
component design 496
component model 327
configuration options 20
constant 88
constructor 67, 76, 105
  example of 83
  prototype property 79
  with Prototype 105
content and behavior
  separating, in widget design 269
Content-Length
  header 222
context menu 275
  disabling 272, 275, 279
context object 103
  pre-binding with Prototype 104
context path 339
contextual help 359
conventions of code generation 30
cookie 392, 406
  library 406
  limits 408
  manipulate 409
  security 409
createContextualFragment() 24
Crockford, Doug 42
cross-browser
  incompatibility 10
  instantiation of XHR 10
cross-field validation 211
cross-server
  proxy 468
  requests 468
cross-server scripting 421
  enabling via server proxy 422
cross-validation group 213
CSS class 138
  as structural annotations 267
  jQuery and 246
CSS. *See* Cascading Style Sheets (CSS)
CSS selector 137, 243
Ctrl+Y key. *See* undo
Ctrl+Z key. *See* undo
cURL 422
custom attributes 205
CustomerManager 397
CustomerServlet 396
customization of browser windows 273
customized undo capability 300

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

D

data
  arrays 393
  caching 392, 394, 402
  client-side 389
  consistency 412
  exchanging 393
  frame 391
  grid 33
  iterating 393
  marshaling 322
  mock 395
  persistence 392
  persisting 406
  posting 218
  sensitive 414
  storing across browser restarts 406
data entry
  errors 359, 366
  validation 203
data marshaling
  across the HTTP interface 8
Date class 216
decoding the POST body 43
defaulting mechanism 79
degradable JavaScript 263
dependency injection 332
deployment descriptor 151, 339, 449
deserialize 44
design, separation of view from behavior 166
desktop
  GUI conventions 240
  navigation techniques and Ajax 247
  programs 337
  UI conventions 247
DHTML 97, 125, 417, 455
  menu 244
dhtmlHistory object 285
dialects, of Ajax 27
Direct Web Remoting (DWR) 150
dirty-checking data 412
disabling
  browser history 272
  navigation, shortcomings of 279
Disc object representation 108
disruptive technology, Ajax as 4
division of responsibility 7
Document Object Model (DOM) 5, 86, 98–99, 134, 137, 477
  Ajax limitations and 50
  API 440
  caching 393
  element 21
  element references 98
  getAttribute() 205
  getElementById 98
  hierarchy 164
  manipulation 9, 29, 347
  NodeList 100
  XML Ajax responses and 45
document ready handler 139
document/literal-style SOAP 62
Dojo 21, 118
  download location 118
  form marashaling 123
  making asynchronous requests 119
dojo.io.bind() 119, 123
dojo.io.Request 21
dojo.js 119
DOM API 347, 462
DOM Level 0 Event Model 165
DOM Level 2 Event Model 165, 169
DOM. *See* Document Object Model (DOM)
domain model 16
domain objects
  over HTTP 44
DomDocument 53
  loading XML into 53
drag and drop 312
  desktop 312
  events 312
  framework 313
  handle 322
  icons 312
  lists 321
  options 331
  region 315
  server communication 315
DragEvent 334
draggable 314
  new 316
  region 316
draggable content
  and back button 272
dragListener 328
dragMask 328
dragOptions 328
dragValue 328
dropdown
  dynamic population 126
  menus and DHTML 243
droppable 314
  add 317
dumb terminal 7
DVD, object representation 111
DWR 150, 322
  debug tools 153
  download location 151
  generated classes 153
  remore procedure calling 151
  servlet 151
DWRUtil 324
Dynamic HTML 5, 9
  revitalization by Ajax 5

E

each() 101, 361
eBay services 464
Echo2 159
eFridge 140
elements, enabling and disabling 92
encapsulation 68, 77
engine.js 324
Enumerable 100
  each() 101
  iterator function 101
errors
  displaying 213
eval() 27, 128, 149, 469
event
  bubbling 165
  capturing 165
  change 185
  filter 320
  hover 318
  models 165
  types 180

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

event handler 89
  example 164
  to trap keyboard input 274
  user interface callbacks 20
event handling
  advanced 169
  basic 165
event models
  Basic 165
  DOM Level 0 165
  DOM Level 2 169
  inline technique 166
  Internet Explorer 170
Event object 172
  browser differences 172
event propagation
  canceling actions 177
  diagrammed 174
  example 176
  stopping 177, 275
Event.observe() 278
events
  beforeunload 187
  blur 182
  bubble handlers 174
  bubble phase 174, 210
  capture handlers 174
  capture phase 174
  change 185
  focus 182
  keyboard 182
  keydown 182
  keyup 182, 209
  load 187
  mouse events 181
  onload versus body script 189
  page events 186
  preventDefault 177
  preventing page unload 188
  propagation 173
  propagation phases 174
  returnValue 177
  target element 173
  target phase 174
  unload 187
  user interface 314
Event.stop() 278
execution stack 406
extend() 107
extending objects 80

F

F5 key 272
faces-config 332
feedback 203, 337
  showing progress 345
  via animation 343
  via page banner 342
fieldset 340
file browser 312
Firefox 10, 165
firstChild 48
Flash 409
  persisting data 392
Flickr 454, 496
  API key 454, 470
  getting photo URLs 459
  NSID 455, 470
  obtaining set IDs 472
  organizer interface 472
  photo services 467
  photo sets 471–472
  photo sizes 460
  REST API 455, 488
  secret numbers 460
  sets 471, 488
  thumbnails 460
  uploading 471
  URL formats 460
focus, keyboard event 182
for loop 101
form
  change detection 227
  data marashaling 123
  disabled elements 231
  element focus 375
  POST 223
  preventing submission 119, 418
  submission 123
  submission hijacking 220
  submit 218
  validation 203, 366
form elements
  name vs. id 99
  obtaning value of 99
form submission
  preventing 177, 195
  via Ajax 195
frame
  data 390
  invisible 390
framework 18
  Ajax 14
  caching 393
Front Controller 339
Function 103
function context 70
  setting 73
function literal 69
functions 68
  as callbacks 70
  as first-class objects 68
  as literals 69
  as method 70
  as object properties 70
  as references 69
  call() method 72–73
  Closures 74
  context 70
  declaring 69
  invocation 71
  invoking with call() 103
  this reference 71

G

Garrett, Jesse James 4
generics 427
Geocoding 430
GET 11
  HTTP method 120, 127
GET requests 42
getAllResponseHeaders() 15
getAttribute() 48, 205
getElementsByTagName() 48
getResponseHeader() 15
GIF animation 344
Global Positioning System (GPS) 417, 421, 471
global variables, avoiding 359
GMail 4
  browser navigation and 279
Google 443
  classification and 236
  license key 444, 448
  search 443
Google Maps 4
  browser navigation and 279
Google Search API 444
Google Suggest 4
googleapi.jar 444

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

GPS. *See* Global Positioning System (GPS)
graphical user interface (GUI) 164
  toolkit 263
grids 247
GUI. *See* graphical user interface (GUI)

H

hash 84
  event 283
Hash class 102
  toQueryString() 102
Hello World 16, 27
help 359
helper object 17, 27
  encapsulating solutions in 17
hierarchical categories 236
high-level API, to reduce coupling 33
history object, JavaScript 280
history state
  and Ajax 289
history-aware tree widget 285
hover() method
  in jQuery 246
hoverclass 317
HTML 134, 137, 164, 421, 492
  button 82
  forms, preventing submission 418
  fragments 24
  specification 337
  table elements and innerHTML 24
  tag 449
  tree-like structure 267
  using custom attributes 205
HTML DOM 86, 99, 440, 458
  name versus id 99
HTTP
  GET 222
  headers 18
  method 11, 18
  protocol 9
  query string 40
  requests 9, 218
  verbs 20
HttpClient 425, 430
HttpServletRequest.getParameter() 43
hybrid models of navigation 259
hyperlinks
  forms, and 7
  HTML forms and 9

I

ICEfaces 326
id, assign 326
iguana 72, 469
impatient users 345
incremental updates, influence on UI design 247
inheritance 80
initialize() 105, 115
innerHTML 21–22, 24, 319, 347, 379, 438, 457, 485
  gotchas 24
  issues with 440, 459
insertAdjacentHTML() 24
Insertion objects
  Prototype library 24
instance methods 81
Interactive 13
interchange format 34
interface, speeding up 392
Internet Explorer 10, 129
  Event Model 165, 170
  select drawing issue 386
  XHR limitations 354
Internet Explorer 7 387
Internet Explorer Event Model 165
inversion of control 332
iPhoto 472
iterator function 101, 485, 490

J

Java 5 generics 427
Java Server Pages 16
Java web application 339
JavaBean 156, 327
java.io.Reader 43
JavaScript 9, 158
  arrays 100
  complexity 8
  constructors 76
  degradable 263
  dependency checking 473
  eval() 128
  functions 68
  hand-written 31
  inheritance 107
  JSON and 42
  JSON-parsing capabilities 37
  methods 77
  object 66
  object orientation 65
  object oriented 76
  package naming 417
  prototypes 79
  scoping rules 75
  tiers 8
  timers 352
  turning off 263
  window object 71
JavaScript function, as callback to XHR 11
JavaScript object 20
JavaScript Object Notation (JSON) 34, 74, 114, 120, 127, 147, 394–396, 399, 406–407, 409–412, 421, 447, 454, 469
  advantages 469
  array 396, 399
  converting Java to 395
  curly braces 35
  drawbacks 469
  example 147
  introduction to 34
  Java and 42
  POST 223
  serialization 394
  square braces 35
  stringify() 36
  XML versus 469
JavaServer Faces (JSF) 326
JavaServer Pages (JSP) 120, 127, 132, 134, 144, 468
jQuery 21, 218, 223, 243–244
  $() 137
  $.ajax() 149
  $.each() 246
  $.get() 145, 149
  $.getJSON() 146, 149
  $.post() 149, 231
  Ajax and 140

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

jQuery *(continued)*
ajaxForm() 225
append() 146
Arrays and 246
browser detection and 246
chaining methods 138
change() 142
class methods 145
CSS classes and 246
document ready handler 139
documentation location 140
download location 136
empty() 146
forms plugin 225
get() 148
importing 141
load() 142, 144, 146
making Ajax requests 136
mouseovers 246
noConflict() 140
Prototype and 139
ready handler 230
ready() 139
separation of markup from script 141
utility functions 145
val() 143, 145
wrapper class 137
jquery.js 136
JSF. *See* JavaServer Faces (JSF)
JSON. *See* JavaScript Object Notation (JSON)
json.js library 42
Json-lib 42
JSONObject 44
json.org 36
JSON.parse() 36, 42
JSP. *See* JavaServer Pages (JSP)
JSP Standard Tag Library (JSTL) 128, 132
JSTL. *See* JSP Standard Tag Library (JSTL)
.jws files and Apache Axis 57

K

keyboard shortcuts
capturing 274
trapping 272, 274
keyboard traversal 375
keydown event 275, 279
keypress event 275
key-value pairs 42

L

latency 338, 389
layout management, client-side 391
legend 340
line-of-business applications 6, 254, 259, 270
list, organize 321
listener function 287
logical bookmarks 280
L-systems 244

M

managed beans 332
MapQuest 465
mashup 467, 494
menu 238, 241
merging 107
method 68, 70, 77
class level 81
instance 81
Microsoft Web Outlook 4
migration path 24
from classic web apps to Ajax 24
MIME type 23, 121, 449
and responseXML 49
setting 15
mistakes, detecting in forms 209
MochiKit 21
Model-View-Controller 326
modifier keys 275
mouse events 91, 312
Mozilla 10, 165
mozXPath.js library 55
multiple clicks 358
eliminating 355
multiple update problem 33
multiple-document interface 251, 254
mutex 405

N

National Weather Service 465
native objects
for SOAP 56
navigation 235
bars 237
controls 280
disabling 272
hotkeys, disabling 279
menu 241
toolbars 272
negative testing 97
.NET 9
netcat 218
Netscape 10
net.sf.json package 43
network round-trip 413
new operator 67, 76, 79
NOAA 465
NodeList 100
converting to array 100, 476

O

object 66
constructor 67
detection 129, 171
fundamentals 66
literal 292
property 66
object orientation 65, 204
class hiearchy example 108
encapsulation 68, 77
extending classes 107
inheritance 80, 107
methods 68
subclassing 208
with Prototype 105
object trees, storing 411
Object-Oriented JavaScript 66
objects, merging with Prototype 107
Observer Pattern 168
onblur 367, 370, 374
onchange 120, 126, 142
onclick 12, 166, 357, 382, 450, 491
onComplete 20

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

oncontextmenu 275
onDrop 314
onfocus 367, 374
onHover 314
onkeydown 379
onload 139
onmouseover 12
onreadystatechange 13
onsubmit 119, 341, 357, 418
open APIs 467
  motivations 416
open source 118
Open Web Application Security Project 414
OpenRico 259
  Accordion control 259
  qooxdoo and 263
  UI development 263
Opera 10
operators
  dot 67
  fetch 67
  indexing 67
  new 67, 76, 79
  square brackets 67
Outlook bar 259

P

page-ranking algorithms 236
parsing
  HTTP response 13
  JSON on the client 39
partition() 269
patterns
  adapter 137
  wrapper 137
performance 389
Perl 422
persistence, Microsoft 392
persisting client data
  log out 392
  refresh 391
phone number, server lookup 119
photos 454
  public and private 455
  sharing 454
PHP 9, 120, 134, 144, 422, 468
portable 314
POST 11, 203
  body 18
  emulating 218, 220
  faking 208
  HTTP method 120
  message format 218
  method 222
  request 42
  RPC 223
post generic data 220
pre-Ajax architecture 7
pre-caching 402
pre-fetching 402
prerendering 393
presentation tier 7
proactive help 337, 359
ProgrammableWeb 465
progress bar 345, 351
property 66
Prototype 97, 125, 139, 178, 204, 266, 397, 432, 467
  $$() 200
  $() 98, 261
  A$() 100
  Ajax.PeriodicalUpdater 134
  Ajax.Request 125, 127
  Ajax.Updater 131
  array extensions 100
  automatic element updating 131
  bind() 104
  checking for 473
  Class class 105
  download location 98, 125
  each() 101, 485
  Enumerable 476
  Enumerable class 100
  event handling 178
  Event namespace API 179
  Event.element 179–180
  Event.findElement 180
  Event.isLeftClick 180
  Event.observe 178–179
  Event.pointerX 180
  Event.pointerY 180
  event-registration mechanism 278
  Event.stop 180
  Event.stopObserving 179
  Event.unloadCache 179
  extend() 107
  F$() 99
  finding elements by CSS selector 200
  Hash class 102, 432
  importing 126, 132
  merging objects 107
  periodic element updating 134
  posting 218
  serialize 192, 197
  strip() 479
  toQueryString() 102
  using with jQuery 139
Prototype library 15
  including 19
prototype property 79
prototype.js 98, 125–126, 316
Prototypes array functions 269
PrototypestopObserving 178
proxy agent 422
Publisher/Subscriber Pattern 168

Q

qooxdoo 248, 254
  development style 250
  initializing 248
  OpenRico and 263
  QxImage 249, 253, 256
  QxTabView 248
  QxTabViewButton 248
  QxTabViewPage 248
  QxToolBar 252
  QxToolBarButton 252
  QxTree 256
  QxTreeFile 256
  QxTreeFolder 256
  QxWindow 252
  root-level container 249
  tab view 248
  toolbar 250
  web design and 250
  windows 250
qooxdoo tree widget
  building 258
  customizing 258
query string 12, 124
  from Hash 102

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

R

readyState 13, 18
Really Simple History 272, 283
redo stack 295
refresh 272
  avoiding 226
  prevention 276
refresh button
  and Ajax 272
relational database 21
Remote Procedure Calling (RPC) 150
removeEventListener 170
Representational State Transfer 421
request
  body 20, 219
  headers 219
  object 320
  parameters 23
reserved characters 102
response 132, 136
responseText 14
  to read generated JavaScript 29
responseXML 14
  MIME type, and 49
  property of XHR object 48
REST 421, 431, 436, 454
  defined 421
reuse 209
Reverse Ajax 326
revert 314
RFID tags 140
Rico 125, 159, 313
Rico Accordion 250, 259
  defining structure of 261
Rigby, Nic 244
right-click menu 275
round-trip 39
RPC. *See* Remote Procedure Calling (RPC)
RSS syndication feeds 45
Ruby on Rails 125, 159
  suitability for JSON 44

S

Safari 10, 165
  and XSLT 50
Safe ActiveX scripting 11
Sajax 159
Sarissa 27, 51, 55, 159
  and XHR object 53
scoping rules 75
Scriptaculous 125, 159
script.aculo.us 313
  serialized lists 325
scriptaculous.js 316
search engine 443
secure socket
  and basic authentication 12
select element, dynamic population 126
selectNodes() 54
selectSingleNode() 54
semantic events 177
send() method 12
separation of concerns 473
serialize 44
server
  access time 413
  date and time display 132
  decrease load 413
  lag 338
  load 136
  POST 220
  resources, preserving 226
  round-trip 401
  saving resources 401
server-side
  maintenance of application history 292
  state persistence versus client-side 293
  undo and server load 309
  undo stack 306
servlet 120, 128, 151, 158, 422
  mapping 151
  path 339
setInterval() 348
setTimeout() 352
shopping cart 33, 314
sidebars 237
Simple Object Access Protocol (SOAP) 45, 56, 421, 443
  messages 56
  nodes, in JavaScript 62
single-page applications 272
snap back 318
SOAP. *See* Simple Object Access Protocol (SOAP)
SOAP-RPC 56
software, reuse 209
stacking order 381, 386
state
  capturing 281
  client 390
  management 284, 388
  server 289
status message 318
status property 14
stopPropagation 177
String object 80
structured data 21
structured objects, communicating over the network 44
Struts 203
subcategories, in categorization schemes 236
subclass 107
success stories of DHTML 243
superclass 107
Swing and qooxdoo 248
synchronous requests 12

T

tabIndex 379
tabs 238
tagging, as classification mechanism 236
text/javascript MIME type 29
The Weather Channel 417
thick client 6
  and Ajax 259
  architecture 34
ThinkCAP JX 159
this 71, 76, 103
  reference 399
threading approximation 405
thumbnail 468, 490
tight coupling 29
timer 348
timestamp 413
Tomcat 429
  web server 16
toolbar 247
  creating in qooxdoo 253
  removing 272

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

Toolkits
  Dojo 118
  DWR 150
  Prototype 125
toQueryString() 102
tree widget 238, 247, 254
  Ajax enabling 259
  and the Web 254
  assembling as HTML 267
  enabling back button 284
Trip-o-matic 467

U

undecorated window 274
undefined 79
undo 272
  handling 293
  stack 295, 300
  system 294
  when to provide 294
undo actions
  bidirectionality of 294
undo/redo
  and Ajax 293, 300
  filtering out duplicates 298
  generic functionality of 295
Uninitialized 13
unnecessary complexity, minimizing 372
updateContentFromNode() 54
URL 11, 123
  encoding 102, 123
  hash as the bookmark ID 283
  servlet mapping 151
usability 337
  and Ajax 6
  challenges 337
  feedback 338, 345
  immediate feedback 209
  maintaining interest 468
  proactive help 359
  showing progress 345
  stacking order 381
  validation 366
user interface feedback 203
user interface mechanisms, for categorization 237
user privacy and server-side history 293
user sessions, preserving app history in 289
util.js 324
UUID. *See* Universally Unique Identifier (UUID) 394

V

validation 158, 190, 366
  classical approach 190
  client versus server 208, 366
  client-side 203
  cross-field 211
  framework 204, 367
  framework implementation 207
  implementing class 207
  input 203
  instant 209
  numeric 208
  performance 203
  reporting 366
  server-assisted 191
variables, global 390
vertical list menu 241
visual feedback 338

W

weather 443
Web 2.0 337
web application
  architecture 7
  deploy and maintain 6
  public perception of 4
web development community 97
web latency 338
Web Service Description Language 58
Web Services client toolkit 27
Webmonkey 406
web.xml 449
  See also deployment descriptor
wget 422
widget structure, defining with CSS 267
window object 71
  setInterval() 134, 352
  setTimeout() 134, 352
window, opening with JavaScript 272
window.location.href 282
window.onload 20
window.open() 273
  customization 273
wizard 391
  IP address 407
workflow 5
  and Ajax 6
workflow logic 270
work-wait pattern 5
World Wide Web Consortium (W3C) 129, 169
wrapper 137
WS.Call object 62
WS.QName object 62
ws-wsajax library 56

X

XHR object
  and Sarissa 53
XHR. *See* XMLHttpRequest (XHR)
XML 15, 44, 421, 469
  advantages 469
  CDATA 477
  collecting body text 477
  comments 477
  design example 470
  digesting 473, 475–476, 490
  drawbacks 469
  JSON versus 469
  parsing issues 470
  POST 223
XML document 100, 446
  NodeList 100
XMLHttpRequest (XHR) 4, 8, 44, 50, 52, 56, 121, 127, 131, 351
  abort() method 14
  callback functions 11
  creating 9
  getAllResponseHeaders() method 15
  getResponseHeader() method 15
  open() method 11
  posting 218
  responseXML property 48

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

XMLHttpRequest (XHR) *(continued)*
- send() method 12
- setRequestHeader() method 15
- SOAP and 56
- status property 14
- XML and 44

XML-RPC 45
XPath 50
- as replacement for DOM methods 50

XSL stylesheet 54
XSLT
- and XPath 55
- stylesheets 50

XSLTProcessor object 54

Y

Yahoo! and categorization 236
Yahoo! Developer Network 416
- application key 416, 432
- location 416

Yahoo! Maps 417, 467, 493, 496
- Geocoding API 421, 430
- importing 418
- panning and zooming 419
- YGeoPoint 487
- YGeoPoint class 419
- YMap class 419

Yahoo! Traffic API 436

Z

z-index 381, 386
zip code 431
zvon.org 55

downloaded from: lib.ommolketab.ir

downloaded from: lib.ommolketab.ir

WEB PROGRAMMING

Ajax IN PRACTICE

Dave Crane, Bear Bibeault, and Jord Sonneveld
with Ted Goddard, Chris Gray, Ram Venkataraman, and Joe Walker

Collectively, web developers have learned a huge amount about Ajax. But it's difficult for any one of them to access and distill all that knowledge. Fortunately, it's now unnecessary: this book collects the most valuable techniques developed by the Ajax community and puts them in your hands.

Ajax in Practice gives you 60 best-practice Ajax techniques illustrated with crisp examples and tons of well-explained code you can reuse. All this is presented in an easy-to-follow, repeating format. The book starts by covering the prerequisites—key Ajax frameworks and object-oriented JavaScript (something you'll need if you want to write scalable Ajax code). Then, it helps you master practical methods for event handling, validation, and state management. A thorough discussion makes each example clear and shows how individual techniques can be combined and extended.

"A 'second-generation' book that distills experience-based practices. Confident and balanced!"

—Ernest J. Friedman-Hill
Sandia National Laboratory
Author of *Jess in Action*

"Any Ajax coder will benefit. [This book] will be useful for years to come."

—Curt Christianson
Microsoft MVP

You'll learn how to

- Implement drag and drop the right way (Chapter 9)
- Control the propagation of an event through the DOM tree (Chapter 5)
- Add back-button and undo support (Chapter 8)
- Implement effective navigation strategies (Chapter 7)
- Prefetch data to improve performance (Chapter 11)
- Build a Yahoo! Maps + Flickr mashup (Chapter 13)

Ajax in Practice brings together a team of experts including **Dave Crane**, leading Ajax authority and best-selling author of Manning's *Ajax in Action*, **Bear Bibeault** of Works.com and JavaRanch, **Jord Sonneveld** of Google, **Chris Gray** of Infor, **Ram Venkataraman** of JBoss, **Ted Goddard** of IceFaces, and **Joe Walker**, creator of DWR.

For more information, code samples, and to purchase an ebook visit www.manning.com/AjaxinPractice

ISBN-10: 1-932394-99-0
ISBN-13: 978-1-932394-99-3

$44.99/Can$ 58.99

downloaded from: lib.ommolketab.ir