Web Services for the Real World



RESTful Web Services

Leonard Richardson & Sam Ruby
Foreword by David Heinemeier Hansson

O'REILLY®

Spotify USA Inc. Exhibit 1014 Page 1

RESTful Web Services

RESTful Web Services

Leonard Richardson and Sam Ruby

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

Spotify USA Inc. Exhibit 1014 Page 3

RESTful Web Services
by Leonard Richardson and Sam Ruby

Copyright © 2007 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

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

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

Editor: Mike Loukides
Copy Editor: Peggy Wallace
Production Editor: Laurel R.T. Ruma
Proofreader: Laurel R.T. Ruma

Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:
May 2007: First Edition

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The vulpine phalanger and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations uses by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

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



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

ISBN-10: 0-596-52926-0
ISBN-13: 978-0-596-52926-0

[M] [06/07]

Table of Contents

Foreword xi

Preface xiii

1. The Programmable Web and Its Inhabitants 1
Kinds of Things on the Programmable Web 4
HTTP: Documents in Envelopes 5
Method Information 8
Scoping Information 11
The Competing Architectures 13
Technologies on the Programmable Web 18
Leftover Terminology 20

2. Writing Web Service Clients 23
Web Services Are Web Sites 23
del.icio.us: The Sample Application 26
Making the Request: HTTP Libraries 29
Processing the Response: XML Parsers 38
JSON Parsers: Handling Serialized Data 44
Clients Made Easy with WADL 47

3. What Makes RESTful Services Different? 49
Introducing the Simple Storage Service 49
Object-Oriented Design of S3 50
Resources 52
HTTP Response Codes 54
An S3 Client 55
Request Signing and Access Control 64
Using the S3 Client Library 70
Clients Made Transparent with ActiveResource 71
Parting Words 77

Spotify USA Inc. Exhibit 1014 Page 5

4. The Resource-Oriented Architecture 79
Resource-Oriented What Now? 79
What's a Resource? 81
URIs 81
Addressability 84
Statelessness 86
Representations 91
Links and Connectedness 94
The Uniform Interface 97
That's It! 105

5. Designing Read-Only Resource-Oriented Services 107
Resource Design 108
Turning Requirements Into Read-Only Resources 109
Figure Out the Data Set 110
Split the Data Set into Resources 112
Name the Resources 117
Design Your Representations 123
Link the Resources to Each Other 135
The HTTP Response 137
Conclusion 140

6. Designing Read/Write Resource-Oriented Services 143
User Accounts as Resources 144
Custom Places 157
A Look Back at the Map Service 165

7. A Service Implementation 167
A Social Bookmarking Web Service 167
Figuring Out the Data Set 168
Resource Design 171
Design the Representation(s) Accepted from the Client 183
Design the Representation(s) Served to the Client 184
Connect Resources to Each Other 185
What's Supposed to Happen? 186
What Might Go Wrong? 187
Controller Code 188
Model Code 205
What Does the Client Need to Know? 209

8. REST and ROA Best Practices 215
Resource-Oriented Basics 215

Spotify USA Inc. Exhibit 1014 Page 6

The Generic ROA Procedure 216
Addressability 216
State and Statelessness 217
Connectedness 218
The Uniform Interface 218
This Stuff Matters 221
Resource Design 227
URI Design 233
Outgoing Representations 234
Incoming Representations 234
Service Versioning 235
Permanent URIs Versus Readable URIs 236
Standard Features of HTTP 237
Faking PUT and DELETE 251
The Trouble with Cookies 252
Why Should a User Trust the HTTP Client? 253

9. The Building Blocks of Services 259
Representation Formats 259
Prepackaged Control Flows 272
Hypermedia Technologies 284

10. The Resource-Oriented Architecture Versus Big Web Services 299
What Problems Are Big Web Services Trying to Solve? 300
SOAP 300
WSDL 304
UDDI 309
Security 310
Reliable Messaging 311
Transactions 312
BPEL, ESB, and SOA 313
Conclusion 314

11. Ajax Applications as REST Clients 315
From AJAX to Ajax 315
The Ajax Architecture 316
A del.icio.us Example 317
The Advantages of Ajax 320
The Disadvantages of Ajax 320
REST Goes Better 322
Making the Request 323
Handling the Response 324
JSON 325

Spotify USA Inc. Exhibit 1014 Page 7

Don't Bogart the Benefits of REST 326
Cross-Browser Issues and Ajax Libraries 327
Subverting the Browser Security Model 331

12. Frameworks for RESTful Services 339
Ruby on Rails 339
Restlet 343
Django 354

A. Some Resources for REST and Some RESTful Resources 365
Standards and Guides 365
Services You Can Use 367

B. The HTTP Response Code Top 42 371
Three to Seven Status Codes: The Bare Minimum 372
1xx: Meta 373
2xx: Success 374
3xx: Redirection 377
4xx: Client-Side Error 380
5xx: Server-Side Error 387

C. The HTTP Header Top Infinity 389
Standard Headers 390
Nonstandard Headers 404

Index 409

Spotify USA Inc. Exhibit 1014 Page 8

Preface

A complex system that works is invariably found to have evolved from a simple system that worked.

—John Gall
Systemantics

We wrote this book to tell you about an amazing new technology. It's here, it's hot, and it promises to radically change the way we write distributed systems. We're talking about the World Wide Web.

Okay, it's not a new technology. It's not as hot as it used to be, and from a technical standpoint it's not incredibly amazing. But everything else is true. In 10 years the Web has changed the way we live, but it's got more change left to give. The Web is a simple, ubiquitous, yet overlooked platform for distributed programming. The goal of this book is to pull out that change and send it off into the world.

It may seem strange to claim that the Web's potential for distributed programming has been overlooked. After all, this book competes for shelf space with any number of other books about web services. The problem is, most of today's "web services" have nothing to do with the Web. In opposition to the Web's simplicity, they espouse a heavyweight architecture for distributed object access, similar to COM or CORBA. Today's "web service" architectures reinvent or ignore every feature that makes the Web successful.

It doesn't have to be that way. We know the technologies behind the Web can drive useful remote services, because those services exist and we use them every day. We know such services can scale to enormous size, because they already do. Consider the Google search engine. What is it but a remote service for querying a massive database and getting back a formatted response? We don't normally think of web sites as "services," because that's programming talk and a web site's ultimate client is a human, but services are what they are.

Every web application—every web site—is a service. You can harness this power for programmable applications if you work with the Web instead of against it, if you don't bury its unique power under layers of abstraction. It's time to put the "web" back into "web services."

Spotify USA Inc. Exhibit 1014 Page 9

The features that make a web site easy for a web surfer to use also make a web service API easy for a programmer to use. To find the principles underlying the design of these services, we can just translate the principles for human-readable web sites into terms that make sense when the surfers are computer programs.

That's what we do in this book. Our goal throughout is to show the power (and, where appropriate, the limitations) of the basic web technologies: the HTTP application protocol, the URI naming standard, and the XML markup language. Our topic is the set of principles underlying the Web: Representational State Transfer, or REST. For the first time, we set down best practices for "RESTful" web services. We cut through the confusion and guesswork, replacing folklore and implicit knowledge with concrete advice.

We introduce the Resource-Oriented Architecture (ROA), a commonsense set of rules for designing RESTful web services. We also show you the view from the client side: how you can write programs to consume RESTful services. Our examples include real-world RESTful services like Amazon's Simple Storage Service (S3), the various incarnations of the Atom Publishing Protocol, and Google Maps. We also take popular services that fall short of RESTfulness, like the del.icio.us social bookmarking API, and rehabilitate them.

The Web Is Simple

Why are we so obsessed with the Web that we think it can do everything? Perhaps we are delusional, the victims of hype. The web is certainly the most-hyped part of the Internet, despite the fact that HTTP is not the most popular Internet protocol. Depending on who's measuring, the bulk of the world's Internet traffic comes from email (thanks to spam) or BitTorrent (thanks to copyright infringement). If the Internet were to disappear tomorrow, email is the application people would miss the most. So why the Web? What makes HTTP, a protocol designed to schlep project notes around a physics lab, also suited for distributed Internet applications?

Actually, to say that HTTP was designed for *anything* is to pay it a pretty big compliment. HTTP and HTML have been called "the Whoopee Cushion and Joy Buzzer of Internet protocols, only comprehensible as elaborate practical jokes"—and that's by someone who *likes* them.* The first version of HTTP sure looked like a joke. Here's a sample interaction between client and server:

Client request	Server response
`GET /hello.txt`	`Hello, world!`

* Clay Shirky, "In Praise of Evolvable Systems" (*http://www.shirky.com/writings/evolve.html*)

Spotify USA Inc. Exhibit 1014 Page 10

That's it. You connected to the server, gave it the path to a document, and then the server sent you the contents of that document. You could do little else with HTTP 0.9. It looked like a featureless rip-off of more sophisticated file transfer protocols like FTP.

This is, surprisingly, a big part of the answer. With tongue only slightly in cheek we can say that HTTP is uniquely well suited to distributed Internet applications because it has no features to speak of. You tell it what you want, and it gives it to you. In a twist straight out of a kung-fu movie,[†] HTTP's weakness is its strength, its simplicity its power.

In that first version of HTTP, cleverly disguised as a lack of features, we can see *addressability* and *statelessness*: the two basic design decisions that made HTTP an improvement on its rivals, and that keep it scalable up to today's mega-sites. Many of the features lacking in HTTP 0.9 have since turned out to be unnecessary or counterproductive. Adding them back actually cripples the Web. Most of the rest were implemented in the 1.0 and 1.1 revisions of the protocol. The other two technologies essential to the success of the Web, URIs and HTML (and, later, XML), are also simple in important senses.

Obviously, these "simple" technologies are powerful enough to give us the Web and the applications we use on it. In this book we go further, and claim that the World Wide Web is a simple and flexible environment for distributed *programming*. We also claim to know the reason for this: that there is no essential difference between the human web designed for our own use, and the "programmable web" designed for consumption by software programs. We say: if the Web is good enough for humans, it's good enough for robots. We just need to make some allowances. Computer programs are good at building and parsing complex data structures, but they're not as flexible as humans when it comes to interpreting documents.

Big Web Services Are Not Simple

There are a number of protocols and standards, mostly built on top of HTTP, designed for building Web Services (note the capitalization). These standards are collectively called the WS-* stack. They include WS-Notification, WS-Security, WSDL, and SOAP. Throughout this book we give the name "Big Web Services" to this collection of technologies as a fairly gentle term of disparagement.

This book does not cover these standards in any great detail. We believe you can implement web services without implementing Big Web Services: that the Web should be all the service you need. We believe the Web's basic technologies are good enough to be considered the default platform for distributed services.

Some of the WS-* standards (such as SOAP) can be used in ways compatible with REST and our Resource-Oriented Architecture. In practice, though, they're used to

[†] *Legend of The Drunken Protocol* (1991)

Spotify USA Inc. Exhibit 1014 Page 11

implement Remote Procedure Call applications over HTTP. Sometimes an RPC style is appropriate, and sometimes other needs take precedence over the virtues of the Web. This is fine.

What we don't like is needless complexity. Too often a programmer or a company brings in Big Web Services for a job that plain old HTTP could handle just fine. The effect is that HTTP is reduced to a transport protocol for an enormous XML payload that explains what's "really" going on. The resulting service is far too complex, impossible to debug, and won't work unless your clients have the exact same setup as you do.

Big Web Services do have one advantage: modern tools can create a web service from your code with a single click, especially if you're developing in Java or C#. If you're using these tools to generate RPC-style web services with the WS-* stack, it probably doesn't matter to you that a RESTful web service would be much simpler. The tools hide all the complexity, so who cares? Bandwidth and CPU are cheap.

This attitude works when you're working in a homogeneous group, providing services behind a firewall for other groups like yours. If your group has enough political clout, you may be able to get people to play your way outside the firewall. But if you want your service to grow to Internet scale, you'll have to handle clients you never planned for, using custom-built software stacks to do things to your service you never imagined were possible. Your users will want to integrate your service with other services you've never heard of. Sound difficult? This already happens on the Web every day.

Abstractions are never perfect. Every new layer creates failure points, interoperability hassles, and scalability problems. New tools can hide complexity, but they can't justify it—and they always add it. Getting a service to work with the Web as a whole means paying attention to adaptability, scalability, and maintainability. Simplicity—that despised virtue of HTTP 0.9—is a prerequisite for all three. The more complex the system, the more difficult it is to fix when something goes wrong.

If you provide RESTful web services, you can spend your complexity on additional features, or on making multiple services interact. Success in providing services also means being part of the Web instead of just "on" the Web: making your information available under the same rules that govern well-designed web sites. The closer you are to the basic web protocols, the easier this is.

The Story of the REST

REST is simple, but it's well defined and not an excuse for implementing web services as half-assed web sites because "they're the same." Unfortunately, until now the main REST reference was chapter five of Roy Fielding's 2000 Ph.D. dissertation, which is a good read for a Ph.D. dissertation, but leaves most of the real-world questions unanswered. ‡ That's because it presents REST not as an architecture but as a way of judging architectures. The term "RESTful" is like the term "object-oriented." A language, a

Spotify USA Inc. Exhibit 1014 Page 12

framework, or an application may be designed in an object-oriented way, but that doesn't make its architecture *the* object-oriented architecture.

Even in object-oriented languages like C++ and Ruby, it's possible to write programs that are not truly object-oriented. HTTP in the abstract does very well on the criteria of REST. (It ought to, since Fielding co-wrote the HTTP standard and wrote his dissertation to describe the architecture of the Web.) But real web sites, web applications, and web services often betray the principles of REST. How can you be sure you're correctly applying the principles to the problem of designing a specific web service?

Most other sources of information on REST are informal: mailing lists, wikis, and weblogs (I list some of the best in Appendix A). Up to now, REST's best practices have been a matter of folklore. What's needed is a concrete architecture based on the REST meta-architecture: a set of simple guidelines for implementing typical services that fulfill the potential of the Web. We present one such architecture in this book as the Resource-Oriented Architecture (see Chapter 4). It's certainly not the only possible high-level RESTful architecture, but we think it's a good one for designing web services that are easy for clients to use.

We wrote the ROA to bring the best practices of web service design out of the realm of folklore. What we've written is a suggested baseline. If you've tried to figure out REST in the past, we hope our architecture gives you confidence that what you're doing is "really" REST. We also hope the ROA will help the community as a whole make faster progress in coming up with and codifying best practices. We want to make it easy for programmers to create distributed web applications that are elegant, that do the job they're designed for, and that participate in the Web instead of merely living on top of it.

We know, however, that it's not enough to have all these technical facts at your disposal. We've both worked in organizations where major architectural decisions didn't go our way. You can't succeed with a RESTful architecture if you never get a chance to use it. In addition to the technical know-how, we must give you the vocabulary to argue for RESTful solutions. We've positioned the ROA as a simple alternative to the RPC-style architecture used by today's SOAP+WSDL services. The RPC architecture exposes internal *algorithms* through a complex programming-language-like interface that's different for every service. The ROA exposes internal *data* through a simple document-processing interface that's always the same. In Chapter 10, we compare the two architectures and show how to argue for the ROA.

‡ Fielding, Roy Thomas. *Architectural Styles and the Design of Network-Based Software Architectures*, Doctoral dissertation, University of California, Irvine, 2000 (*http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*)

Spotify USA Inc. Exhibit 1014 Page 13

Reuniting the Webs

Programmers have been using web sites as web services for years—unofficially, of course.§ It's difficult for a computer to understand web pages designed for human consumption, but that's never stopped hackers from fetching pages with automated clients and screen-scraping the interesting bits. Over time, this drive was sublimated into programmer-friendly technologies for exposing a web site's functionality in officially sanctioned ways—RSS, XML-RPC, and SOAP. These technologies formed a programmable web, one that extended the human web for the convenience of software programs.

Our ultimate goal in this book is to reunite the programmable web with the human web. We envision a single interconnected network: a World Wide Web that runs on one set of servers, uses one set of protocols, and obeys one set of design principles. A network that you can use whether you're serving data to human beings or computer programs.

The Internet and the Web did not have to exist. They come to us courtesy of misallocated defense money, skunkworks engineering projects, worse-is-better engineering practices, big science, naive liberal idealism, cranky libertarian politics, technofetishism, and the sweat and capital of programmers and investors who thought they'd found an easy way to strike it rich.

The result is, amazingly, a simple, open (for now), almost universal platform for networked applications. This platform contains much of human knowledge and supports most fields of human endeavor. We think it's time to seriously start applying its rules to distributed programming, to open up that information and those processes to automatic clients. If you agree, this book will show you to do it.

What's in This Book?

In this book we focus on practical issues: how to design and implement RESTful web services, and clients for those services. Our secondary focus is on theory: what it means to be RESTful, and why web services should be more RESTful instead of less. We don't cover everything, but we try to hit today's big topics, and because this is the first book of its kind, we return to the core issue—how to design a RESTful service—over and over again.

The first three chapters introduce web services from the client's perspective and show what's special about RESTful services.

§ For an early example, see Jon Udell's 1996 *Byte* article "On-Line Componentware" (*http://www.byte.com/art/9611/sec9/art1.htm*). Note: "A powerful capability for ad hoc distributed computing arises naturally from the architecture of the Web." That's from 1996, folks.

Spotify USA Inc. Exhibit 1014 Page 14

Chapter 1, The Programmable Web and Its Inhabitants

In this chapter we introduce web services in general: programs that go over the Web and ask a foreign server to provide data or run an algorithm. We demonstrate the three common web service architectures: RESTful, RPC-style, and REST-RPC hybrid. It shows sample HTTP requests and responses for each architecture, along with typical client code.

Chapter 2, Writing Web Service Clients

In this chapter we show you how to write clients for existing web services, using an HTTP library and an XML parser. We introduce a popular REST-RPC service (the web service for the social bookmarking site del.icio.us) and demonstrate clients written in Ruby, Python, Java, C#, and PHP. We also give technology recommendations for several other languages, without actually showing code. JavaScript and Ajax are covered separately in Chapter 11.

Chapter 3, What Makes RESTful Services Different?

We take the lessons of Chapter 2 and apply them to a purely RESTful service: Amazon's Simple Storage Service (S3). While building an S3 client we illustrate some important principles of REST: resources, representations, and the uniform interface.

The next six chapters form the core of the book. They focus on designing and implementing your own RESTful services.

Chapter 4, The Resource-Oriented Architecture

A formal introduction to REST, not in its abstract form but in the context of a specific architecture for web services. Our architecture is based on four important REST concepts: resources, their names, their representations, and the links between them. Its services should be judged by four RESTful properties: addressability, statelessness, connectedness, and the uniform interface.

Chapter 5, Designing Read-Only Resource-Oriented Services

We present a procedure for turning an idea or a set of requirements into a set of RESTful resources. These resources are read-only: clients can get data from your service but they can't send any data of their own. We illustrate the procedure by designing a web service for serving navigable maps, inspired by the Google Maps web application.

Chapter 6, Designing Read/Write Resource-Oriented Services

We extend the procedure from the previous chapter so that clients can create, modify, and delete resources. We demonstrate by adding two new kinds of resource to the map service: user accounts and user-defined places.

Chapter 7, A Service Implementation

We remodel an RPC-style service (the del.icio.us REST-RPC hybrid we wrote clients for back in Chapter 2) as a purely RESTful service. Then we implement that service as a Ruby on Rails application. Fun for the whole family!

Spotify USA Inc. Exhibit 1014 Page 15

Chapter 8, REST and ROA Best Practices

In this chapter we collect our earlier suggestions for service design into one place, and add new suggestions. We show how standard features of HTTP can help you with common problems and optimizations. We also give resource-oriented designs for tough features like transactions, which you may have thought were impossible to do in RESTful web services.

Chapter 9, The Building Blocks of Services

Here we describe extra technologies that work on top of REST's big three of HTTP, URI, and XML. Some of these technologies are file formats for conveying state, like XHTML and its microformats. Some are hypermedia formats for showing clients the levers of state, like WADL. Some are sets of rules for building RESTful web services, like the Atom Publishing Protocol.

The last three chapters cover specialized topics, each of which could make for a book in its own right:

Chapter 10, The Resource-Oriented Architecture Versus Big Web Services

We compare our architecture, and REST in general, to another leading brand. We think that RESTful web services are simpler, more scalable, easier to use, better attuned to the philosophy of the Web, and better able to handle a wide variety of clients than are services based on SOAP, WSDL, and the WS-* stack.

Chapter 11, Ajax Applications as REST Clients

Here we explain the Ajax architecture for web applications in terms of web services: an Ajax application is just a web service client that runs inside your web browser. That makes this chapter an extension of Chapter 2. We show how to write clients for RESTful web services using `XMLHttpRequest` and the standard JavaScript library.

Chapter 12, Frameworks for RESTful Services

In the final chapter we cover three popular frameworks that make it easy to implement RESTful web services: Ruby on Rails, Restlet (for Java), and Django (for Python).

We also have three appendixes we hope you find useful:

Appendix A, Some Resources for REST and Some RESTful Resources

The first part lists interesting standards, tutorials, and communities related to RESTful web services. The second part lists some existing, public RESTful web services that you can use and learn from.

Appendix B, The HTTP Response Code Top 42

Describes every standard HTTP response code (plus one extension), and explains when you'd use each one in a RESTful web service.

Appendix C, The HTTP Header Top Infinity

Does the same thing for HTTP headers. It covers every standard HTTP header, and a few extension headers that are useful for web services.

Spotify USA Inc. Exhibit 1014 Page 16

Which Parts Should You Read?

We organized this book for the reader who's interested in web services in general: someone who learns by doing, but who doesn't have much experience with web services. If that describes you, the simplest path through this book is the best. You can start at the beginning, read through Chapter 9, and then read onward as you're interested.

If you have more experience, you might take a different path through the book. If you're only concerned with writing clients for existing services, you'll probably focus on Chapters 1, 2, 3, and 11—the sections on service design won't do you much good. If you want to create your own web service, or you're trying to figure out what REST really means, you might start reading from Chapter 3. If you want to compare REST to the WS-* technologies, you might start by reading Chapters 1, 3, 4, and 10.

Administrative Notes

This book has two authors (Leonard and Sam), but for the rest of the book we'll be merging our identities into a single authorial "I." In the final chapter (Chapter 12), the authorial "I" gets a little bit more crowded, as Django and Restlet developers join in to show how their frameworks let you build RESTful services.

We assume that you're a competent programmer, but not that you have any experience with web programming in particular. What we say in this book is not tied to any programming language, and we include sample code for RESTful clients and services in a variety of languages. But whenever we're not demonstrating a specific framework or language, we use Ruby (*http://www.ruby-lang.org/*) as our implementation language.

We chose Ruby because it's concise and easy to read, even for programmers who don't know the language. (And because it's nice and confusing in conjunction with Sam's last name.) Ruby's standard web framework, Ruby on Rails, is also one of the leading implementation platforms for RESTful web services. If you don't know Ruby, don't worry: we include lots of comments explaining Ruby-specific idioms.

The sample programs in this book are available for download from this book's official web site (*http://www.oreilly.com/catalog/9780596529260*). This includes the entire Rails application from Chapter 7, and the corresponding Restlet and Django applications from Chapter 12. It also includes Java implementations of many of the clients that only show up in the book as Ruby implementations. These client programs use the Restlet library, and were written by Restlet developers Jerome Louvel and Dave Pawson. If you're more familiar with Java than with Ruby, these implementations may help you grasp the concepts behind the code. Most notably, there's a full Java implementation of the Amazon S3 client from Chapter 3 in there.

Spotify USA Inc. Exhibit 1014 Page 17

Conventions Used in This Book

The following typographical conventions are used in this book:

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

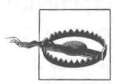
`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

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

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.



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



This icon indicates a warning or caution.

Using Code Examples

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*RESTful Web Services* by Leonard Richardson and Sam Ruby. Copyright 2007 O'Reilly Media, Inc., 978-0-596-52926-0."

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

Spotify USA Inc. Exhibit 1014 Page 18

Safari® Enabled



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

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

How to Contact Us

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

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

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

http://www.oreilly.com/catalog/9780596529260

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

bookquestions@oreilly.com

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

http://www.oreilly.com

Acknowledgments

We're ultimately indebted to the people whose work made us see that we could program directly with HTTP. For Sam, it was Rael Dornfest with his Blosxom blogging application. Leonard's experience stems from building screen-scraping applications in the mid-90s. His thanks go to those whose web design made their sites usable as web services: notably, the pseudonymous author of the online comic "Pokey the Penguin."

Once we had this insight, Roy Fielding was there to flesh it out. His thesis named and defined something that was for us only a feeling. Roy's theoretical foundation is what we've tried to build on.

Spotify USA Inc. Exhibit 1014 Page 19

In writing this book we had an enormous amount of help from the REST community. We're grateful for the feedback we got from Benjamin Carlyle, David Gourley, Joe Gregorio, Marc Hadley, Chuck Hinson, Pete Lacey, Larry Liberto, Benjamin Pollack, Aron Roberts, Richard Walker, and Yohei Yamamoto. Others helped us unknowingly, through their writings: Mark Baker, Tim Berners-Lee, Alex Bunardzic, Duncan Cragg, David Heinemeier Hansson, Ian Hickson, Mark Nottingham, Koranteng Ofosu-Amaah, Uche Ogbuji, Mark Pilgrim, Paul Prescod, Clay Shirky, Brian Totty, and Jon Udell. Of course, all opinions in this book, and any errors and omissions, are our own.

Our editor Michael Loukides was helpful and knowledgeable throughout the process of developing this book. We'd also like to thank Laurel Ruma and everyone else at O'Reilly for their production work.

Finally, Jerome Louvel, Dave Pawson, and Jacob Kaplan-Moss deserve special thanks. Their knowledge of Restlet and Django made Chapter 12 possible.

Spotify USA Inc. Exhibit 1014 Page 20

CHAPTER 1

The Programmable Web and Its Inhabitants

When you write a computer program, you're not limited to the algorithms you can think up. Your language's standard library gives you some algorithms. You can get more from books, or in third-party libraries you find online. Only if you're on the very cutting edge should you have to come up with your own algorithms.

If you're lucky, the same is true for data. Some applications are driven entirely by the data the users type in. Sometimes data just comes to you naturally: if you're analyzing spam, you should have no problem getting all you need. You can download a few public data sets—word lists, geographical data, lists of prime numbers, public domain texts—as though they were third-party libraries. But if you need some other kind of data, it doesn't look good. Where's the data going to come from? More and more often, it's coming from the programmable web.

When you—a human being—want to find a book on a certain topic, you probably point your web browser to the URI of an online library or bookstore: say, *http://www.amazon.com/*.



The common term for the address of something on the Web is "URL." I say "URI" throughout this book because that's what the HTTP standard says. Every URI on the Web is also a URL, so you can substitute "URL" wherever I say "URI" with no loss of meaning.

You're served a web page, a document in HTML format that your browser renders graphically. You visually scan the page for a search form, type your topic (say, "web services") into a text box, and submit the form. At this point your web browser makes a second HTTP request, to a URI that incorporates your topic. To continue the Amazon example, the second URI your browser requests would be something like *http://amazon.com/s?url=search-alias%3Dstripbooks&field-keywords=web+services*.

Spotify USA Inc. Exhibit 1014 Page 21

The web server at amazon.com responds by serving a second document in HTML format. This document contains a description of your search results, links to additional search options, and miscellaneous commercial enticements (see Example 1-1). Again, your browser renders the document in graphical form, and you look at it and decide what to do from there.

Example 1-1. Part of the HTML response from amazon.com

```
...
<a href="http://www.amazon.com/Restful-Web-Services-Leonard-Richardson/dp/...>
 <span class="srTitle">RESTful Web Services</span>
</a>

by Leonard Richardson and Sam Ruby

<span class="bindingBlock">
 (<span class="binding">Paperback</span> - May 1, 2007)
</span>
```

The Web you use is full of data: book information, opinions, prices, arrival times, messages, photographs, and miscellaneous junk. It's full of *services*: search engines, online stores, weblogs, wikis, calculators, and games. Rather than installing all this data and all these programs on your own computer, you install one program—a web browser—and access the data and services through it.

The programmable web is just the same. The main difference is that instead of arranging its data in attractive HTML pages with banner ads and cute pastel logos, the programmable web usually serves stark, brutal XML documents. The programmable web is not neccessarily for human consumption. Its data is intended as input to a software program that does something amazing.

Example 1-2 shows a Ruby script that uses the programmable web to do a traditional human web task: find the titles of books matching a keyword. It hides the web access under a programming language interface, using the Ruby/Amazon library (*http://www.caliban.org/ruby/ruby-amazon.shtml*).

Example 1-2. Searching for books with a Ruby script

```
#!/usr/bin/ruby -w
# amazon-book-search.rb
require 'amazon/search'

if ARGV.size != 2
  puts "Usage: #{$0} [Amazon Web Services AccessKey ID] [text to search for]"
  exit
end
access_key, search_request = ARGV
req = Amazon::Search::Request.new(access_key)
# For every book in the search results...
req.keyword_search(search_request, 'books', Amazon::Search::LIGHT) do |book|
  # Print the book's name and the list of authors.
```


Spotify USA Inc. Exhibit 1014 Page 22

```
    puts %{"#{book.product_name}" by #{book.authors.join(', ')}}
end
```

To run this program, you'll need to sign up for an Amazon Web Services account (*http://aws.amazon.com/*) and customize the Ruby code with your Access Key ID. Here's a sample run of the program:

```
$ ruby amazon-search.rb C1D4NQS41IMK2 "restful web services"
"RESTful Web Services" by Leonard Richardson, Sam Ruby
"Hacking with Ruby: Ruby and Rails for the Real World" by Mark Watson
```

At its best, the programmable web works the same way as the human web. When *amazon-book-search.rb* calls the method `Amazon::Search::Request#keyword_search`, the Ruby program starts acting like a web browser. It makes an HTTP request to a URI: in this case, something like *http://xml.amazon.com/onca/xml3?KeywordSearch=restful+web+services&mode=books&f=xml&type=lite&page=1*. The web server at `xml.amazon.com` responds with an XML document. This document, shown in Example 1-3, describes the search results, just like the HTML document you see in your web browser, but in a more structured form.

Example 1-3. Part of the XML response from xml.amazon.com

```
...
<ProductName>RESTful Web Services</ProductName>
<Catalog>Book</Catalog>
<Authors>
 <Author>Leonard Richardson</Author>
 <Author>Sam Ruby</Author>
</Authors>
<ReleaseDate>01 May, 2007</ReleaseDate>
...
```

Once a web browser has submitted its HTTP request, it has a fairly easy task. It needs to render the response in a way a human being can understand. It doesn't need to figure out what the HTTP response means: that's the human's job. A web service client doesn't have this luxury. It's programmed in advance, so it has to be both the web browser that fetches the data, and the "human" who decides what the data means. Web service clients must automatically extract meaning from HTTP responses and make decisions based on that meaning.

In Example 1-2, the web service client parses the XML document, extracts some interesting information (book titles and authors), and prints that information to standard output. The program *amazon-book-search.rb* is effectively a small, special-purpose web browser, relaying data to a human reader. It could easily do something else with the Amazon book data, something that didn't rely on human intervention at all: stick the book titles into a database, maybe, or use the author information to drive a recommendation engine.

And the data doesn't have to always flow toward the client. Just as you can bend parts of the human web to your will (by posting on your weblog or buying a book), you can

Spotify USA Inc. Exhibit 1014 Page 23

write clients that modify the programmable web. You can use it as a storage space or as another source of algorithms you don't have to write yourself. It depends on what service you need, and whether you can find someone else to provide it.

Example 1-4 is an example of a web service client that modifies the programmable web: the s3sh command shell for Ruby (*http://amazon.rubyforge.org/*). It's one of many clients written against another of Amazon's web services: S3, or the Simple Storage Service (*http://aws.amazon.com/s3*). In Chapter 3 I cover S3's workings in detail, so if you're interested in using s3sh for yourself, you can read up on S3 there.

To understand this s3sh transcript, all you need to know is that Amazon S3 lets its clients store labelled pieces of data ("objects") in labelled containers ("buckets"). The s3sh program builds an interactive programming interface on top of S3. Other clients use S3 as a backup tool or a web host. It's a very flexible service.

Example 1-4. Manipulating the programmable web with s3sh and S3

```
$ s3sh
>> Service.buckets.collect { |b| b.name }
=> ["example.com"]

>> my_bucket = Bucket.find("example.com")

>> contents = open("disk_file.txt").read
=> "This text is the contents of the file disk_file.txt"

>> S3Object.store("mydir/mydocument.txt", contents, my_bucket.name)

>> my_bucket['directory/document.txt'].value
=> "This text is the contents of the file disk_file.txt"
```

In this chapter I survey the current state of the programmable web. What technologies are being used, what architectures are they used to implement, and what design styles are the most popular? I show some real code and some real HTTP conversations, but my main goal in this chapter is to get you thinking about the World Wide Web as a way of connecting computer programs to each other, on the same terms as it connects human beings to each other.

Kinds of Things on the Programmable Web

The programmable web is based on HTTP and XML. Some parts of it serve HTML, JavaScript Object Notation (JSON), plain text, or binary documents, but most parts use XML. And it's all based on HTTP: if you don't use HTTP, you're not on the web.* Beyond that small island of agreement there is little but controversy. The terminology isn't set, and different people use common terms (like "REST," the topic of this book) in ways that combine into a vague and confusing mess. What's missing is a coherent way of classifying the programmable web. With that in place, the meanings of individual terms will become clear.

Spotify USA Inc. Exhibit 1014 Page 24

Imagine the programmable web as an ecosystem, like the ocean, containing many kinds of strange creatures. Ancient scientists and sailors classified sea creatures by their superficial appearance: whales were lumped in with the fish. Modern scientists classify animals according to their position in the evolutionary tree of all life: whales are now grouped with the other mammals. There are two analogous ways of classifying the services that inhabit the programmable web: by the technologies they use (URIs, SOAP, XML-RPC, and so on), or by the underlying architectures and design philosophies.

Usually the two systems for classifying sea creatures get along. You don't need to do DNA tests to know that a tuna is more like a grouper than a sea anenome. But if you really want to understand why whales can't breathe underwater, you need to stop classifying them as fish (by superficial appearance) and start classifying them as mammals (by underlying architecture).†

When it comes to classifying the programmable web, most of today's terminology sorts services by their superficial appearances: the technologies they use. These classifications work in most cases, but they're conceptually lacking and they lead to whale-fish mistakes. I'm going to present a taxonomy based on architecture, which shows how technology choices follow from underlying design principles. I'm exposing divisions I'll come back to throughout the book, but my main purpose is to zoom in on the parts of the programmable web that can reasonably be associated with the term "REST."

HTTP: Documents in Envelopes

If I was classifying marine animals I'd start by talking about the things they have in common: DNA, cellular structure, the laws of embryonic development. Then I'd show how animals distinguish themselves from each other by specializing away from the common ground. To classify the programmable web, I'd like to start off with an overview of HTTP, the protocol that all web services have in common.

HTTP is a document-based protocol, in which the client puts a document in an envelope and sends it to the server. The server returns the favor by putting a response document in an envelope and sending it to the client. HTTP has strict standards for what the envelopes should look like, but it doesn't much care what goes inside. Example 1-5 shows a sample envelope: the HTTP request my web browser sends when I

* Thanks to Big Web Services' WS-Addressing standard, it's now possible to create a web service that's not on the Web: one that uses email or TCP as its transport protocol instead of HTTP. I don't think absolutely everything has to be on the Web, but it does seem like you should have to call this bizarre spectacle something other than a web service. This point isn't really important, since in practice nearly everyone uses HTTP. Thus the footnote. The only exceptions I know of are eBay's web services, which can send you SOAP documents over email as well as HTTP.

† Melville, in *Moby-Dick*, spends much of Chapter 22 ("Cetology") arguing that the whale is a fish. This sounds silly but he's not denying that whales have lungs and give milk; he's arguing for a definition of "fish" based on appearance, as opposed to Linnaeus's definition "from the law of nature" (*ex lege naturae*).

Spotify USA Inc. Exhibit 1014 Page 25

visit the homepage of `oreilly.com`. I've truncated two lines to make the text fit on the printed page.

Example 1-5. An HTTP GET request for http://www.oreilly.com/index.html

```
GET /index.html HTTP/1.1
Host: www.oreilly.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.12)...
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,...
Accept-Language: us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-15,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

In case you're not familiar with HTTP, now is a good time to point out the major parts of the HTTP request. I use these terms throughout the book.

The HTTP method
: In this request, the method is "GET." In other discussions of REST you may see this called the "HTTP verb" or "HTTP action."

  The name of the HTTP method is like a method name in a programming language: it indicates how the client expects the server to process this envelope. In this case, the client (my web browser) is trying to GET some information from the server (`www.oreilly.com`).

The path
: This is the portion of the URI to the right of the hostname: here, *http://www.oreilly.com/index.html* becomes "/index.html." In terms of the envelope metaphor, the path is the address on the envelope. In this book I sometimes refer to the "URI" as shorthand for just the path.

The request headers
: These are bits of metadata: key-value pairs that act like informational stickers slapped onto the envelope. This request has eight headers: `Host`, `User-Agent`, `Accept`, and so on. There's a standard list of HTTP headers (see Appendix C), and applications can define their own.

The entity-body, also called the document or representation
: This is the document that inside the envelope. This particular request has no entity-body, which means the envelope is empty! This is typical for a GET request, where all the information needed to complete the request is in the path and the headers.

The HTTP response is also a document in a envelope. It's almost identical in form to the HTTP request. Example 1-6 shows a trimmed version of what the server at `oreilly.com` sends my web browser when I make the request in Example 1-5.

Example 1-6. The response to an HTTP GET request for http://www.oreilly.com/index.html

```
HTTP/1.1 200 OK
Date: Fri, 17 Nov 2006 15:36:32 GMT
```


Spotify USA Inc. Exhibit 1014 Page 26

```
Server: Apache
Last-Modified: Fri, 17 Nov 2006 09:05:32 GMT
Etag: "7359b7-a7fa-455d8264
Accept-Ranges: bytes
Content-Length: 43302
Content-Type: text/html
X-Cache: MISS from www.oreilly.com
Keep-Alive: timeout=15, max=1000
Connection: Keep-Alive

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
...
<title>oreilly.com -- Welcome to O'Reilly Media, Inc.</title>
...
```

The response can be divided into three parts:

The HTTP response code
: This numeric code tells the client whether its request went well or poorly, and how the client should regard this envelope and its contents. In this case the GET operation must have succeeded, since the response code is 200 ("OK"). I describe the HTTP response codes in Appendix B.

The response headers
: Just as with the request headers, these are informational stickers slapped onto the envelope. This response has 11 headers: `Date`, `Server`, and so on.

The entity-body or representation
: Again, this is the document inside the envelope, and this time there actually is one! The entity-body is the fulfillment of my GET request. The rest of the response is just an envelope with stickers on it, telling the web browser how to deal with the document.

  The most important of these stickers is worth mentioning separately. The response header `Content-Type` gives the *media type* of the entity-body. In this case, the media type is `text/html`. This lets my web browser know it can render the entity-body as an HTML document: a web page.

  There's a standard list of media types (*http://www.iana.org/assignments/media-types/*). The most common media types designate textual documents (`text/html`), structured data documents (`application/xml`), and images (`image/jpeg`). In other discussions of REST or HTTP, you may see the media type called the "MIME type," "content type," or "data type."

Spotify USA Inc. Exhibit 1014 Page 27

Method Information

HTTP is the one thing that all "animals" on the programmable web have in common. Now I'll show you how web services distinguish themselves from each other. There are two big questions that today's web services answer differently. If you know how a web service answers these questions, you'll have a good idea of how well it works with the Web.

The first question is how the client can convey its intentions to the server. How does the server know a certain request is a request to retrieve some data, instead of a request to delete that same data or to overwrite it with different data? Why should the server do *this* instead of doing *that*?

I call the information about what to do with the data the *method information*. One way to convey method information in a web service is to put it in the HTTP method. Since this is how RESTful web services do it, I'll have a lot more to say about this later. For now, note that the five most common HTTP methods are GET, HEAD, PUT, DELETE, and POST. This is enough to distinguish between "retrieve some data" (GET), "delete that same data" (DELETE), and "overwrite it with different data" (PUT).

The great advantage of HTTP method names is that they're standardized. Of course, the space of HTTP method names is much more limited than the space of method names in a programming language. Some web services prefer to look for application-specific method names elsewhere in the HTTP request: usually in the URI path or the request document.

Example 1-7 is a client for a web service that keeps its method information in the path: the web service for Flickr, Yahoo!'s online photo-sharing application. This sample application searches Flickr for photos. To run this program, you'll need to create a Flickr account and apply for an API key (*http://www.flickr.com/services/api/keys/apply/*).

Example 1-7. Searching Flickr for pictures

```
#!/usr/bin/ruby -w
# flickr-photo-search.rb
require 'open-uri'
require 'rexml/document'

# Returns the URI to a small version of a Flickr photo.
def small_photo_uri(photo)
  server = photo.attribute('server')
  id = photo.attribute('id')
  secret = photo.attribute('secret')
  return "http://static.flickr.com/#{server}/#{id}_#{secret}_m.jpg"
end

# Searches Flickr for photos matching a certain tag, and prints a URI
# for each search result.
def print_each_photo(api_key, tag)
  # Build the URI
  uri = "http://www.flickr.com/services/rest?method=flickr.photos.search" +
```


Spotify USA Inc. Exhibit 1014 Page 28

```
    "&api_key=#{api_key}&tags=#{tag}"

  # Make the HTTP request and get the entity-body.
  response = open(uri).read

  # Parse the entity-body as an XML document.
  doc = REXML::Document.new(response)

  # For each photo found...
  REXML::XPath.each(doc, '//photo') do |photo|
    # ...generate and print its URI
    puts small_photo_uri(photo) if photo
  end
end

# Main program
#
if ARGV.size < 2
  puts "Usage: #{$0} [Flickr API key] [search term]"
  exit
end

api_key, tag = ARGV
print_each_photo(api_key, tag)
```

XPath: The Bluffer's Guide

XPath is a domain-specific language for slicing up XML documents without writing a lot of code. It has many intimidating features, but you can get by with just a little bit of knowledge. The key is to think of an XPath expression as a rule for extracting tags or other elements from an XML document. There aren't many XPath expressions in this book, but I'll explain every one I use.

To turn an XPath expression into English, read it from right to left. The expression `//photo` means:

Find every photo tag	photo
no matter where it is in the document.	//

The Ruby code `REXML::XPath.each(doc, '//photo')` is a cheap way to iterate over every `photo` tag without having to traverse the XML tree.

This program makes HTTP requests to URIs like *http://www.flickr.com/services/rest?method=flickr.photos.search&api_key=xxx&tag=penguins*. How does the server know what the client is trying to do? Well, the method name is pretty clearly `flickr.photos.search`. Except: the HTTP method is GET, and I am *get*ting information, so it might be that the `method` thing is a red herring. Maybe the method information really goes in the HTTP action.

Spotify USA Inc. Exhibit 1014 Page 29

This hypothesis doesn't last for very long, because the Flickr API supports many methods, not just "get"-type methods such as `flickr.photos.search` and `flickr.people.findByEmail`, but also methods like `flickr.photos.addTags`, `flickr.photos.comments.deleteComment`, and so on. All of them are invoked with an HTTP GET request, regardless of whether or not they "get" any data. It's pretty clear that Flickr is sticking the method information in the `method` query variable, and expecting the client to ignore what the HTTP method says.

By contrast, a typical SOAP service keeps its method information in the entity-body and in a HTTP header. Example 1-8 is a Ruby script that searches the Web using Google's SOAP-based API.

Example 1-8. Searching the Web with Google's search service

```
#!/usr/bin/ruby -w
# google-search.rb
require 'soap/wsdlDriver'

# Do a Google search and print out the title of each search result
def print_page_titles(license_key, query)
  wsdl_uri = 'http://api.google.com/GoogleSearch.wsdl'
  driver = SOAP::WSDLDriverFactory.new(wsdl_uri).create_rpc_driver
  result_set = driver.doGoogleSearch(license_key, query, 0, 10, true, ' ',
                                     false, ' ', ' ', ' ')
  result_set.resultElements.each { |result| puts result.title }
end

# Main program.
if ARGV.size < 2
  puts "Usage: #{$0} [Google license key] [query]"
  exit
end

license_key, query = ARGV
print_page_titles(license_key, query)
```



While I was writing this book, Google announced that it was deprecating its SOAP search service in favor of a RESTful, resource-oriented service (which, unfortunately, is encumbered by legal restrictions on use in a way the SOAP service isn't). I haven't changed the example because Google's SOAP service still makes the best example I know of, and because I don't expect you to actually run this program. I just want you to look at the code, and the SOAP and WSDL documents the code relies on.

OK, that probably wasn't very informative, because the WSDL library hides most of the details. Here's what happens. When you call the `doGoogleSearch` method, the WSDL library makes a POST request to the "endpoint" of the Google SOAP service, located at the URI *http://api.google.com/search/beta2*. This single URI is the destination for every API call, and only POST requests are ever made to it. All of these details are in

Spotify USA Inc. Exhibit 1014 Page 30

the WSDL file found at *http://api.google.com/GoogleSearch.wsdl*, which contains details like the definition of `doGoogleSearch` (Example 1-9).

Example 1-9. Part of the WSDL description for Google's search service

```
<operation name="doGoogleSearch">
 <input message="typens:doGoogleSearch"/>
 <output message="typens:doGoogleSearchResponse"/>
</operation>
```

Since the URI and the HTTP method never vary, the method information—that "doGoogleSearch"—can't go in either place. Instead, it goes into the entity-body of the POST request. Example 1-10 shows what HTTP request you might make to do a search for REST.

Example 1-10. A sample SOAP RPC call

```
POST search/beta2 HTTP/1.1
Host: api.google.com
Content-Type: application/soap+xml
SOAPAction: urn:GoogleSearchAction

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <gs:doGoogleSearch xmlns:gs="urn:GoogleSearch">
   <q>REST</q>
   ...
  </gs:doGoogleSearch>
 </soap:Body>
</soap:Envelope>
```

The method information is "doGoogleSearch." That's the name of the XML tag inside the SOAP `Envelope`, it's the name of the `operation` in the WSDL file, and it's the name of the Ruby method in Example 1-8. It's also found in the value of the `SOAPAction` HTTP request header: some SOAP implementations look for it there instead of inside the entity-body.

Let's bring things full circle by considering not the Google SOAP search API, but the Google search engine itself. To use your web browser to search Google's data set for REST, you'd send a GET request to *http://www.google.com/search?q=REST* and get an HTML response back. The method information is kept in the HTTP method: you're GETting a list of search results.

Scoping Information

The other big question web services answer differently is how the client tells the server which part of the data set to operate on. Given that the server understands that the client wants to (say) delete some data, how can it know which data the client wants to delete? Why should the server operate on *this* data instead of *that* data?

Spotify USA Inc. Exhibit 1014 Page 31

I call this information the *scoping information*. One obvious place to put it is in the URI path. That's what most web sites do. Think once again about a search engine URI like *http://www.google.com/search?q=REST*. There, the method information is "GET," and the scoping information is "/search?q=REST." The client is trying to GET a list of search results about REST, as opposed to trying to GET something else: say, a list of search results about jellyfish (the scoping information for that would be "/search?q=jellyfish"), or the Google home page (that would be "/").

Many web services put scoping information in the path. Flickr's is one: most of the query variables in a Flickr API URI are scoping information. `tags=penguin` scopes the `flickr.photos.search` method so it only searches for photos tagged with "penguin." In a service where the method information defines a method in the programming language sense, the scoping information can be seen as a set of arguments to that method. You could reasonably expect to see `flickr.photos.search(tags=penguin)` as a line of code in some programming language.

The alternative is to put the scoping information into the entity-body. A typical SOAP web service does it this way. Example 1-10 contains a `q` tag whose contents are the string "REST." That's the scoping information, nestled conveniently inside the `doGoogleSearch` tag that provides the method information.

The service design determines what information is method information and what's scoping information. This is most obvious in cases like Flickr and Google, where the web site and the web service do the same thing but have different designs. These two URIs contain the same information:

- *http://flickr.com/photos/tags/penguin*
- *http://api.flickr.com/services/rest/?method=flickr.photos.search&tags=penguin*

In the first URI, the method information is "GET" and the scoping information is "photos tagged 'penguin.'" In the second URI, the method information is "do a photo search" and the scoping information is "penguin." From a technical standpoint, there's no difference between the two: both of them use HTTP GET. The differences only become apparent at the level of architecture, when you take a step back and notice values for `methodname` like `flickr.photos.delete`, which take HTTP's GET method into places it wasn't meant to go.

Another example: in the Google SOAP API, the fact that you're doing a search is method information (`doGoogleSearch`). The search query is scoping information (`q`). On the Google web site, both "search" and the value for "q" are scoping information. The method information is HTTP's standard GET. (If the Google SOAP API offered a method called `doGoogleSearchForREST`, it would be defining the method information so expansively that you'd need no scoping information to do a search for REST.)

Spotify USA Inc. Exhibit 1014 Page 32

The Competing Architectures

Now that I've identified the two main questions that web services answer differently, I can group web services by their answers to the questions. In my studies I've identified three common web service architectures: RESTful resource-oriented, RPC-style, and REST-RPC hybrid. I'll cover each in turn.

RESTful, Resource-Oriented Architectures

The main topic of this book is the web service architectures which can be considered RESTful: those which get a good score when judged on the criteria set forth in Roy Fielding's dissertation. Now, lots of architectures are technically RESTful,‡ but I want to focus on the architectures that are best for web services. So when I talk about RESTful web services, I mean services that look like the Web. I'm calling this kind of service *resource-oriented*. In Chapter 3 I'll introduce the basic concepts of resource-oriented REST, in the context of a real web service: Amazon's Simple Storage Service. Starting in Chapter 5, I'll talk you through the defining characteristics of REST, and define a good architecture for RESTful web services: the Resource-Oriented Architecture.

In RESTful architectures, the method information goes into the HTTP method. In Resource-Oriented Architectures, the scoping information goes into the URI. The combination is powerful. Given the first line of an HTTP request to a resource-oriented RESTful web service ("GET /reports/open-bugs HTTP/1.1"), you should understand basically what the client wants to do. The rest of the request is just details; indeed, you can make many requests using only one line of HTTP. If the HTTP method doesn't match the method information, the service isn't RESTful. If the scoping information isn't in the URI, the service isn't resource-oriented. These aren't the *only* requirements, but they're good rules of thumb.

A few well-known examples of RESTful, resource-oriented web services include:

- Services that expose the Atom Publishing Protocol (*http://www.ietf.org/html.charters/atompub-charter.html*) and its variants such as GData (*http://code.google.com/apis/gdata/*)
- Amazon's Simple Storage Service (S3) (*http://aws.amazon.com/s3*)
- Most of Yahoo!'s web services (*http://developer.yahoo.com/*)
- Most other read-only web services that don't use SOAP
- Static web sites
- Many web applications, especially read-only ones like search engines

‡ More than you'd think. The Google SOAP API for web search technically has a RESTful architecture. So do many other read-only SOAP and XML-RPC services. But these are bad architectures for web services, because they look nothing like the Web.

Spotify USA Inc. Exhibit 1014 Page 33

Whenever I cover unRESTful architectures, as well as architectures that aren't resource-oriented, I do it with some ulterior motive. In this chapter, I want to put RESTful web services into perspective, against the larger backdrop of the programmable web. In Chapter 2, I'm widening the book's coverage of real web services, and showing that you can use the same client tools whether or not a service exactly fits my preferred architecture. In Chapter 10, I'm making an argument in a long-running debate about what the programmable web *should* look like.

RPC-Style Architectures

An RPC-style web service accepts an envelope full of data from its client, and sends a similar envelope back. The method and the scoping information are kept inside the envelope, or on stickers applied to the envelope. What *kind* of envelope is not important to my classification, but HTTP is a popular envelope format, since any web service worthy of the name must use HTTP anyway. SOAP is another popular envelope format (transmitting a SOAP document over HTTP puts the SOAP envelope inside an HTTP envelope). Every RPC-style service defines a brand new vocabulary. Computer programs work this way as well: every time you write a program, you define functions with different names. By contrast, all RESTful web services share a standard vocabulary of HTTP methods. Every object in a RESTful service responds to the same basic interface.

The XML-RPC protocol for web services is the most obvious example of the RPC architecture. XML-RPC is mostly a legacy protocol these days, but I'm going to start off with it because it's relatively simple and easy to explain. Example 1-11 shows a Ruby client for an XML-RPC service that lets you look up anything with a Universal Product Code.

Example 1-11. An XML-RPC example: looking up a product by UPC

```
#!/usr/bin/ruby -w
# xmlrpc-upc.rb

require 'xmlrpc/client'
def find_product(upc)
  server = XMLRPC::Client.new2('http://www.upcdatabase.com/rpc')
  begin
    response = server.call('lookupUPC', upc)
  rescue XMLRPC::FaultException => e
    puts "Error: "
    puts e.faultCode
    puts e.faultString
  end
end

puts find_product("001441000055")['description']
# "Trader Joe's Thai Rice Noodles"
```

An XML-RPC service models a programming language like C. You call a function (`lookupUPC`) with some arguments ("001441000055") and get a return value back. The

Spotify USA Inc. Exhibit 1014 Page 34

method data (the function name) and the scoping data (the arguments) are put inside an XML document. Example 1-12 gives a sample document.

Example 1-12. An XML document describing an XML-RPC request

```
<?xml version="1.0" ?>
 <methodCall>
  <methodName>lookupUPC</methodName>
  <params>
   <param><value><string>001441000055</string></value></param>
  </params>
 </methodCall>
```

This XML document is put into an envelope for transfer to the server. The envelope is an HTTP request with a method, URI, and headers (see Example 1-13). The XML document becomes the entity-body inside the HTTP envelope.

Example 1-13. An HTTP envelope that contains an XML document which describes an XML-RPC request

```
POST /rpc HTTP/1.1
Host: www.upcdatabase.com
User-Agent: XMLRPC::Client (Ruby 1.8.4)
Content-Type: text/xml; charset=utf-8
Content-Length: 158
Connection: keep-alive

<?xml version="1.0" ?>
<methodCall>
 <methodName>lookupUPC</methodName>
 ...
</methodCall>
```

The XML document changes depending on which method you're calling, but the HTTP envelope is always the same. No matter what you do with the UPC database service, the URI is always *http://www.upcdatabase.com/rpc* and the HTTP method is always POST. Simply put, an XML-RPC service ignores most features of HTTP. It exposes only one URI (the "endpoint"), and supports only one method on that URI (POST).

Where a RESTful service would expose different URIs for different values of the scoping information, an RPC-style service typically exposes a URI for each "document processor": something that can open the envelopes and transform them into software commands. For purposes of comparison, Example 1-14 shows what that code might look like if the UPC database were a RESTful web service.

Example 1-14. A hypothetical code sample: a RESTful UPC lookup service

```
require 'open-uri'
upc_data = open('http://www.upcdatabase.com/upc/00598491').read()
...
```


Spotify USA Inc. Exhibit 1014 Page 35

Here, the method information is contained in the HTTP method. The default HTTP method is GET, which is equivalent in this scenario to `lookupUPC`. The scoping information is contained in the URI. The hypothetical service exposes an enormous number of URIs: one for every possible UPC. By contrast, the HTTP envelope is empty: an HTTP GET request contains no entity-body at all.

For another example of a client for an RPC-style service, look back at Example 1-8. Google's SOAP search API is an RPC-style service that uses SOAP as its envelope format.

A service that uses HTTP POST heavily or exclusively is probably an RPC-style service. Again, this isn't a sure sign, but it's a tip-off that the service isn't very interested in putting its method information in the HTTP method. An otherwise RESTful service that uses HTTP POST a lot tends to move toward a REST-RPC hybrid architecture.

A few well-known examples of RPC-style web services:

- All services that use XML-RPC
- Just about every SOAP service (see the "Technologies on the Programmable Web" section later in this chapter for a defense of this controversial statement)
- A few web applications (generally poorly designed ones)

REST-RPC Hybrid Architectures

This is a term I made up for describing web services that fit somewhere in between the RESTful web services and the purely RPC-style services. These services are often created by programmers who know a lot about real-world web applications, but not much about the theory of REST.

Take another look at this URI used by the Flickr web service: *http://www.flickr.com/services/rest?api_key=xxx&method=flickr.photos.search&tags=penguin*. Despite the "rest" in the URI, this was clearly designed as an RPC-style service, one that uses HTTP as its envelope format. It's got the scoping information ("photos tagged 'penguin'") in the URI, just like a RESTful resource-oriented service. But the method information ("search for photos") also goes in the URI. In a RESTful service, the method information would go into the HTTP method (GET), and whatever was leftover would become scoping information. As it is, this service is simply using HTTP as an envelope format, sticking the method and scoping information wherever it pleases. This is an RPC-style service. Case closed.

Except...look at Example 1-15.

Example 1-15. A sample HTTP request to the Flickr web service

```
GET services/rest?api_key=xxx&method=flickr.photos.search&tags=penguin HTTP/1.1
Host: www.flickr.com
```

That's the HTTP request a client makes when remotely calling this procedure. Now it looks like the method information is in the HTTP method. I'm sending a GET request

Spotify USA Inc. Exhibit 1014 Page 36

to get something. What am I getting? A list of search results for photos tagged "penguin." What used to look like method information ("photoSearch()") now looks like scoping information ("photos/tag/penguin"). Now the web service looks RESTful.

This optical illusion happens when an RPC-style service uses plain old HTTP as its envelope format, and when both the method and the scoping information happen to live into the URI portion of the HTTP request. If the HTTP method is GET, and the point of the web service request is to "get" information, it's hard to tell whether the method information is in the HTTP method or in the URI. Look at the HTTP requests that go across the wire and you see the requests you'd see for a RESTful web service. They may contain elements like "method=flickr.photos.search" but that could be interpreted as scoping information, the way "photos/" and "search/" are scoping information. These RPC-style services have elements of RESTful web services, more or less by accident. They're only using HTTP as a convenient envelope format, but they're using it in a way that overlaps with what a RESTful service might do.

Many read-only web services qualify as entirely RESTful and resource-oriented, even though they were designed in the RPC style! But if the service allows clients to write to the data set, there will be times when the client uses an HTTP method that doesn't match up with the true method information. This keeps the service from being as RESTful as it could be. Services like these are the ones I consider to be REST-RPC hybrids.

Here's one example. The Flickr web API asks clients to use HTTP GET even when they want to modify the data set. To delete a photo you make a GET request to a URI that includes `method=flickr.photos.delete`. That's just not what GET is for, as I'll show in "Split the Data Set into Resources [115]. The Flickr web API is a REST-RPC hybrid: RESTful when the client is retrieving data through GET, RPC-style when the client is modifying the data set.

A few well-known examples of REST-RPC hybrid services include:

- The del.icio.us API
- The "REST" Flickr web API
- Many other allegedly RESTful web services
- Most web applications

From a design standpoint, I don't think anybody sets out to to design a service as a REST-RPC hybrid. Because of the way HTTP works, any RPC-style service that uses plain HTTP and exposes multiple URIs tends to end up either RESTful or hybrid. Many programmers design web services exactly as they'd design web applications, and end up with hybrid services.

The existence of hybrid architectures has caused a lot of confusion. The style comes naturally to people who've designed web applications, and it's often claimed that hybrid architectures are RESTful: after all, they work "the same way" as the human web. A lot of time has been spent trying to distinguish RESTful web services from these

Spotify USA Inc. Exhibit 1014 Page 37

mysterious others. My classification of the "others" as REST-RPC hybrids is just the latest in a long line of neologisms. I think this particular neologism is the most accurate and useful way to look at these common but baffling services. If you've encountered other ways of describing them ("HTTP+POX" is the most popular at the time of writing), you might want read on, where I explain those other phrases in terms of what I'm saying in this book.

The Human Web Is on the Programmable Web

In the previous sections I claimed that all static web sites are RESTful. I claimed that web applications fall into one of the three categories, the majority being REST-RPC hybrids. Since the human web is made entirely of static web sites and web applications, this means that the entire human web is also on the programmable web! By now this should not be surprising to you. A web browser is a software program that makes HTTP requests and processes the responses somehow (by showing them to a human). That's exactly what a web service client is. *If it's on the Web, it's a web service.*

My goal in this book is not to make the programmable web bigger. That's almost impossible: the programmable web already encompasses nearly everything with an HTTP interface. My goal is to help make the programmable web *better*: more uniform, better-structured, and using the features of HTTP to greatest advantage.

Technologies on the Programmable Web

I've classified web services by their underlying architectures, distinguishing the fish from the whales. Now I can examine the technologies they use, without confusing technology and architecture.

HTTP

All web services use HTTP, but they use it in different ways. A request to a RESTful web services puts the method information in the HTTP method and the scoping information in the URI. RPC-style web services tend to ignore the HTTP method, looking for method and scoping information in the URI, HTTP headers, or entity-body. Some RPC-style web services use HTTP as an envelope containing a document, and others only use it as an unlabelled envelope containing another envelope.

URI

Again, all web services use URIs, but in different ways. What I'm about to say is a generalization, but a fairly accurate one. A RESTful, resource-oriented service exposes a URI for every piece of data the client might want to operate on. A REST-RPC hybrid exposes a URI for every operation the client might perform: one URI to fetch a piece of data, a different URI to delete that same data. An RPC-style service exposes one URI

Spotify USA Inc. Exhibit 1014 Page 38

for every processes capable of handling Remote Procedure Calls (RPC). There's usually only one such URI: the service "endpoint."

XML-RPC

A few, mostly legacy, web services use XML-RPC on top of HTTP. XML-RPC is a data structure format for representing function calls and their return values. As the name implies, it's explicitly designed to use an RPC style.

SOAP

Lots of web services use SOAP on top of HTTP. SOAP is an envelope format, like HTTP, but it's an XML-based envelope format.

Now I'm going to say something controversial. *To a first approximation, every current web service that uses SOAP also has an RPC architecture.* This is controversial because many SOAP programmers think the RPC architecture is déclassé and prefer to call their services "message-oriented" or "document-oriented" services.

Well, all web services are message-oriented, because HTTP itself is message-oriented. An HTTP request is just a message: an envelope with a document inside. The question is what that document says. SOAP-based services ask the client to stick a second envelope (a SOAP document) inside the HTTP envelope. Again, the real question is what it says inside the envelope. A SOAP envelope can contain any XML data, just as an HTTP envelope can contain any data in its entity-body. But in every *existing* SOAP service, the SOAP envelope contains a description of an RPC call in a format similar to that of XML-RPC.

There are various ways of shuffling this RPC description around and giving it different labels—"document/literal" or "wrapped/literal"—but any way you slice it, you have a service with a large vocabulary of method information, a service that looks for scoping information inside the document rather than on the envelope. These are defining features of the RPC architecture.

I emphasize that this is not a fact about SOAP, just a fact about how it's currently used. SOAP, like HTTP, is just a way of putting data in an envelope. Right now, though, the only data that ever gets put in that envelope is XML-RPC-esque data about how to call a remote function, or what's the return value from such a function. I argue this point in more detail in Chapter 10.

WS-*

These standards define special XML "stickers" for the SOAP envelope. The stickers are analagous to HTTP headers.

Spotify USA Inc. Exhibit 1014 Page 39

WSDL

The Web Service Description Language (WSDL) is an XML vocabulary used to describe SOAP-based web services. A client can load a WSDL file and know exactly which RPC-style methods it can call, what arguments those methods expect, and which data types they return. Nearly every SOAP service in existence exposes a WSDL file, and most SOAP services would be very difficult to use without their WSDL files serving as guides. As I discuss in Chapter 10, WSDL bears more responsiblity than any other technology for maintaining SOAP's association with the RPC style.

WADL

The Web Application Description Language (WADL) is an XML vocabulary used to describe RESTful web services. As with WSDL, a generic client can load a WADL file and be immediately equipped to access the full functionality of the corresponding web service. I discuss WADL in Chapter 9.

Since RESTful services have simpler interfaces, WADL is not nearly as neccessary to these services as WSDL is to RPC-style SOAP services. This is a good thing, since as of the time of writing there are few real web services providing official WADL files. Yahoo!'s web search service is one that does.

Leftover Terminology

Believe it not, there are some common terms used in discussions of REST that I haven't mentioned yet. I haven't mentioned them because I think they're inaccurate or entirely outside the scope of this book. But I owe you explanations of *why* I think this, so you can decide whether or not you agree. Feel free to skip this section if you haven't heard these terms.

Service-Oriented Architecture
: This is a big industry buzzword. I'm not going to dwell on it for two reasons. First, the term is not very well defined. Second, to the extent that it is defined, it means something like: "a software architecture based on the production and consumption of web services." In this book I talk about the design of individual services. A book on service-oriented architecture should work on a slightly higher level, showing how to use services as software components, how to integrate them into a coherent whole. I don't cover that sort of thing in this book.

SOAP as a competitor to REST
: If you get involved with web service debates you'll hear this one a lot. You won't hear it here because it gives the wrong impression. The primary competitors to RESTful architectures are RPC architectures, not specific technologies like SOAP. It is true that basically every SOAP service that now exists has an RPC architecture, but SOAP is just a way of putting a document in an envelope with stickers on it,

Spotify USA Inc. Exhibit 1014 Page 40

like HTTP. SOAP is tied to the RPC architecture mainly by historical contingency and the current generation of automated tools.

There is a real tension here, but it's not one I'll cover much in this book. Roughly speaking, it's the tension between services that put their documents in a SOAP envelope and then an HTTP envelope; and services that only use the HTTP envelope.

HTTP+POX

Stands for HTTP plus Plain Old XML. This term covers roughly those services I call REST-RPC hybrid services. They overlap with RESTful designs, especially when it comes to retrieving data, but their basic architecture is RPC-oriented.

I don't like this term because Plain Old XML is inaccurate. The interesting thing about these services is not that they produce plain old XML documents (as opposed to XML documents wrapped in SOAP envelopes). Some of these services don't serve XML at all: they serve JSON, plain text, or binary files. No, the interesting thing about these services is their RPC architecture. *That's* what puts them in opposition to REST.

STREST

Means Service-Trampled REST. This is another term for REST-RPC hybrid architectures. It's more accurate than HTTP+POX since it conveys the notion of a RESTful architecture taken over by something else: in this case, the RPC style.

This is a cute acronym but I don't like it, because it buys into a myth that the only true web services are RPC-style services. After all, the service that trampled your REST was an RPC service. If you think that REST services are real services, it doesn't make sense to cry "Help! I had some REST but then this *Service* got into it!" RPC-Trampled REST would be more accurate, but that's a lousy acronym.

High and low REST

Yet another way of distinguishing between truly RESTful services and the ones I call REST-RPC hybrids. High REST services are just those that adhere closely to the Fielding dissertation. Among other things, they put method information in the HTTP method and scoping information in the URI. Low REST services are presumed to have deviated. Since low REST services tend to deviate from orthodoxy in a particular direction (toward the RPC style), I prefer a more specific terminology.

Spotify USA Inc. Exhibit 1014 Page 41

CHAPTER 3

What Makes RESTful Services Different?

I pulled a kind of bait-and-switch on you earlier, and it's time to make things right. Though this is a book about RESTful web services, most of the real services I've shown you are REST-RPC hybrids like the del.icio.us API: services that don't quite work like the rest of the Web. This is because right now, there just aren't many well-known RESTful services that work like the Web. In previous chapters I wanted to show you clients for real services you might have heard of, so I had to take what I could get.

The del.icio.us and Flickr APIs are good examples of hybrid services. They work like the Web when you're fetching data, but they're RPC-style services when it comes time to modify the data. The various Yahoo! search services are very RESTful, but they're so simple that they don't make good examples. The Amazon E-Commerce Service (seen in Example 1-2) is also quite simple, and defects the RPC style on a few obscure but important points.

These services are all useful. I think the RPC style is the wrong one for web services, but that never prevents me from writing an RPC-style client if there's interesting data on the other side. I can't use Flickr or the del.icio.us API as examples of how to *design* RESTful web services, though. That's why I covered them early in the book, when the only thing I was trying to show was what's on the programmable web and how to write HTTP clients. Now that we're approaching a heavy design chapter, I need to show you what a service looks like when it's RESTful and resource-oriented.

Introducing the Simple Storage Service

Two popular web services can answer this call: the Atom Publishing Protocol (APP), and Amazon's Simple Storage Service (S3). (Appendix A lists some publicly deployed RESTful web services, many of which you may not have heard of.) The APP is less an actual service than a set of instructions for building a service, so I'm going to start with S3, which actually exists at a specific place on the Web. In Chapter 9 I discuss the APP,

Spotify USA Inc. Exhibit 1014 Page 42

Atom, and related topics like Google's GData. For much of the rest of this chapter, I'll explore S3.

S3 is a way of storing any data you like, structured however you like. You can keep your data private, or make it accessible by anyone with a web browser or BitTorrent client. Amazon hosts the storage and the bandwidth, and charges you by the gigabyte for both. To use the example S3 code in this chapter, you'll need to sign up for the S3 service by going to *http://aws.amazon.com/s3*. The S3 technical documentation is at *http://docs.amazonwebservices.com/AmazonS3/2006-03-01/*.

There are two main uses for S3, as a:

Backup server
: You store your data through S3 and don't give anyone else access to it. Rather than buying your own backup disks, you're renting disk space from Amazon.

Data host
: You store your data on S3 and give others access to it. Amazon serves your data through HTTP or BitTorrent. Rather than paying an ISP for bandwidth, you're paying Amazon. Depending on your existing bandwidth costs this can save you a lot of money. Many of today's web startups use S3 to serve data files.

Unlike the services I've shown so far, S3 is not inspired by any existing web site. The del.icio.us API is based on the del.icio.us web site, and the Yahoo! search services are based on corresponding web sites, but there's no web page on *amazon.com* where you fill out HTML forms to upload your files to S3. S3 is intended only for programmatic use. (Of course, if you use S3 as a data host, people will use it through their web browsers, without even knowing they're making a web service call. It'll act like a normal web site.)

Amazon provides sample libraries for Ruby, Python, Java, C#, and Perl (see *http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=47*). There are also third-party libraries, like Ruby's AWS::S3 (*http://amazon.rubyforge.org/*), which includes the s3sh shell I demonstrated back in Example 1-4.

Object-Oriented Design of S3

S3 is based on two concepts: S3 "buckets" and S3 "objects." An object is a named piece of data with some accompanying metadata. A bucket is a named container for objects. A bucket is analogous to the filesystem on your hard drive, and an object to one of the files on that filesystem. It's tempting to compare a bucket to a *directory* on a filesystem, but filesystem directories can be nested and buckets can't. If you want a directory structure inside your bucket, you need to simulate one by giving your objects names like "directory/subdirectory/file-object."

Spotify USA Inc. Exhibit 1014 Page 43

A Few Words About Buckets

A bucket has one piece of information associated with it: the name. A bucket name can only contain the characters A through Z, a through z, 0 through 9, underscore, period, and dash. I recommend staying away from uppercase letters in bucket names.

As I mentioned above, buckets cannot contain other buckets: only objects. Each S3 user is limited to 100 buckets, and your bucket name cannot conflict with anyone else's. I recommend you either keep everything in one bucket, or name each bucket after one of your projects or domain names.

A Few Words About Objects

An object has four parts to it:

- A reference to the parent bucket.
- The data stored in that object (S3 calls this the "value").
- A name (S3 calls it the "key").
- A set of metadata key-value pairs associated with the object. This is mostly custom metadata, but it may also include values for the standard HTTP headers `Content-Type` and `Content-Disposition`.

If I wanted to host the O'Reilly web site on S3, I'd create a bucket called "oreilly.com," and fill it with objects whose keys were "" (the empty string), "catalog," "catalog/9780596529260," and so on. These objects correspond to the URIs *http://oreilly.com/*, *http://oreilly.com/catalog*, and so on. The object's values would be the HTML contents of O'Reilly's web pages. These S3 objects would have their `Content-Type` metadata value set to `text/html`, so that people browsing the site would be served these objects as HTML documents, as opposed to XML or plain text.

What If S3 Was a Standalone Library?

If S3 was implemented as an object-oriented code library instead of a web service, you'd have two classes `S3Bucket` and `S3Object`. They'd have getter and setter methods for their data members: `S3Bucket#name`, `Object.value=`, `S3Bucket#addObject`, and the like. The `S3Bucket` class would have an instance method `S3Bucket#getObjects` that returned a list of `S3Object` instances, and a class method `S3Bucket.getBuckets` that returned all of your buckets. Example 3-1 shows what the Ruby code for this class might look like.

Example 3-1. S3 implemented as a hypothetical Ruby library

```
class S3Bucket
  # A class method to fetch all of your buckets.
  def self.getBuckets
  end

  # An instance method to fetch the objects in a bucket.
```


Spotify USA Inc. Exhibit 1014 Page 44

```
  def getObjects
  end
  ...
end

class S3Object
  # Fetch the data associated with this object.
  def data
  end

  # Set the data associated with this object.
  def data=(new_value)
  end
  ...
end
```

Resources

Amazon exposes S3 as two different web services: a RESTful service based on plain HTTP envelopes, and an RPC-style service based on SOAP envelopes. The RPC-style service exposes functions much like the methods in Example 3-1's hypothetical Ruby library: `ListAllMyBuckets`, `CreateBucket`, and so on. Indeed, many RPC-style web services are automatically generated from their implementation methods, and expose the same interfaces as the programming-language code they call behind the scenes. This works because most modern programming (including object-oriented programming) is procedural.

The RESTful S3 service exposes all the functionality of the RPC-style service, but instead of doing it with custom-named functions, it exposes standard HTTP objects called *resources*. Instead of responding to custom method names like `getObjects`, a resource responds to one or more of the six standard HTTP methods: GET, HEAD, POST, PUT, DELETE, and OPTIONS.

The RESTful S3 service provides three types of resources. Here they are, with sample URIs for each:

- The list of your buckets (`https://s3.amazonaws.com/`). There's only one resource of this type.
- A particular bucket (`https://s3.amazonaws.com/`*`{name-of-bucket}`*`/`). There can be up to 100 resources of this type.
- A particular S3 object inside a bucket (`https://s3.amazonaws.com/`*`{name-of-bucket}/{name-of-object}`*). There can be infinitely many resources of this type.

Each method from my hypothetical object-oriented S3 library corresponds to one of the six standard methods on one of these three types of resources. The getter method `S3Object#name` corresponds to a GET request on an "S3 object" resource, and the setter method `S3Object#value=` corresponds to a PUT request on the same resource. Factory

Spotify USA Inc. Exhibit 1014 Page 45

methods like `S3Bucket.getBuckets` and relational methods like `S3Bucket#getObjects` correspond to GET methods on the "bucket list" and "bucket" resources.

Every resource exposes the same interface and works the same way. To get an object's value you send a GET request to that object's URI. To get only the metadata for an object you send a HEAD request to the same URI. To create a bucket, you send a PUT request to a URI that incorporates the name of the bucket. To add an object to a bucket, you send PUT to a URI that incorporates the bucket name and object name. To delete a bucket or an object, you send a DELETE request to its URI.

The S3 designers didn't just make this up. According to the HTTP standard this is what GET, HEAD, PUT, and DELETE are *for*. These four methods (plus POST and OPTIONS, which S3 doesn't use) suffice to describe *all* interaction with resources on the Web. To expose your programs as web services, you don't need to invent new vocabularies or smuggle method names into URIs, or do anything except think carefully about your resource design. Every REST web service, no matter how complex, supports the same basic operations. All the complexity lives in the resources.

Table 3-1 shows what happens when you send an HTTP request to the URI of an S3 resource.

Table 3-1. S3 resources and their methods

	GET	HEAD	PUT	DELETE
The bucket list (/)	List your buckets	-	-	-
A bucket (/*{bucket}*)	List the bucket's objects	-	Create the bucket	Delete the bucket
An object (/*{bucket}*/*{object}*)	Get the object's value and metadata	Get the object's metadata	Set the object's value and metadata	Delete the object

That table looks kind of ridiculous. Why did I take up valuable space by printing it? Everything just does what it says. And *that* is why I printed it. In a well-designed RESTful service, *everything does what it says*.

You may well be skeptical of this claim, given the evidence so far. S3 is a pretty generic service. If all you're doing is sticking data into named slots, then of course you can implement the service using only generic verbs like GET and PUT. In Chapter 5 and Chapter 6 I'll show you strategies for mapping *any* kind of action to the uniform interface. For a sample preconvincing, note that I was able to get rid of `S3Bucket.getBuckets` by defining a new resource as "the list of buckets," which responds only to GET. Also note that `S3Bucket#addObject` simply disappeared as a natural consequence of the resource design, which requires that every object be associated with some bucket.

Compare this to S3's RPC-style SOAP interface. To get the bucket list through SOAP, the method name is `ListAllMyBuckets`. To get the contents of a bucket, the method

Spotify USA Inc. Exhibit 1014 Page 46

name is `ListBucket`. With the RESTful interface, it's always GET. In a RESTful service, the URI designates an object (in the object-oriented sense) and the method names are standardized. The same few methods work the same way across resources and services.

HTTP Response Codes

Another defining feature of a RESTful architecture is its use of HTTP response codes. If you send a request to S3, and S3 handles it with no problem, you'll probably get back an HTTP response code of 200 ("OK"), just like when you successfully fetch a web page in your browser. If something goes wrong, the response code will be in the 3xx, 4xx, or 5xx range: for instance, 500 ("Internal Server Error"). An error response code is a signal to the client that the metadata and entity-body should not be interpreted as a response to the request. It's not what the client asked for: it's the server's attempt to tell the client about a problem. Since the response code isn't part of the document or the metadata, the client can see whether or not an error occurred just by looking at the first three bytes of the response.

Example 3-2 shows a sample error response. I made an HTTP request for an object that didn't exist (`https://s3.amazonaws.com/crummy.com/nonexistent/object`). The response code is 404 ("Not Found").

Example 3-2. A sample error response from S3

```
404 Not Found
Content-Type: application/xml
Date: Fri, 10 Nov 2006 20:04:45 GMT
Server: AmazonS3
Transfer-Encoding: chunked
X-amz-id-2: /sBIPQxHJCsyRXJwGWNzxuL5P+K96/Wvx4FhvVACbjRfNbhbDyBH5RC511sIzOwO
X-amz-request-id: ED2168503ABB7BF4

<?xml version="1.0" encoding="UTF-8"?>
<Error>
 <Code>NoSuchKey</Code>
 <Message>The specified key does not exist.</Message>
 <Key>nonexistent/object</Key>
 <RequestId>ED2168503ABB7BF4</RequestId>
 <HostId>/sBIPQxHJCsyRXJwGWNzxuL5P+K96/Wvx4FhvVACbjRfNbhbDyBH5RC511sIzOwO</HostId>
</Error>
```

HTTP response codes are underused on the human web. Your browser doesn't show you the HTTP response code when you request a page, because who wants to look at a numeric code when you can just look at the document to see whether something went wrong? When an error occurs in a web application, most web applications send 200 ("OK") along with a human-readable document that talks about the error. There's very little chance a human will mistake the error document for the document they requested.

On the programmable web, it's just the opposite. Computer programs are good at taking different paths based on the value of a numeric variable, and very bad at figuring

Spotify USA Inc. Exhibit 1014 Page 47

out what a document "means." In the absence of prearranged rules, there's no way for a program to tell whether an XML document contains data or describes an error. HTTP response codes are the rules: rough conventions about how the client should approach an HTTP response. Because they're not part of the entity-body or metadata, a client can understand what happened even if it has no clue how to read the response.

S3 uses a variety of response codes in addition to 200 ("OK") and 404 ("Not Found"). The most common is probably 403 ("Forbidden"), used when the client makes a request without providing the right credentials. S3 also uses a few others, including 400 ("Bad Request"), which indicates that the server couldn't understand the data the client sent; and 409 ("Conflict"), sent when the client tries to delete a bucket that's not empty. For a full list, see the S3 technical documentation under "The REST Error Response." I describe *every* HTTP response code in Appendix B, with a focus on their application to web services. There are 39 official HTTP response codes, but only about 10 are important in everyday use.

An S3 Client

The Amazon sample libraries, and the third-party contributions like AWS::S3, eliminate much of the need for custom S3 client libraries. But I'm not telling you about S3 just so you'll know about a useful web service. I want to use it to illustrate the theory behind REST. So I'm going to write a Ruby S3 client of my own, and dissect it for you as I go along.

Just to show it can be done, my library will implement an object-oriented interface, like the one from Example 3-1, on top of the S3 service. The result will look like ActiveRecord or some other object-relational mapper. Instead of making SQL calls under the covers to store data in a database, though, it'll make HTTP requests under the covers to store data on the S3 service. Rather than give my methods resource-specific names like `getBuckets` and `getObjects`, I'll try to use names that reflect the underlying RESTful interface: `get`, `put`, and so on.

The first thing I need is an interface to Amazon's rather unusual web service authorization mechanism. But that's not as interesting as seeing the web service in action, so I'm going to skip it for now. I'm going to create a very small Ruby module called `S3::Authorized`, just so my other `S3` classes can include it. I'll come back to it at the end, and fill in the details.

Example 3-3 shows a bit of throat-clearing code.

Example 3-3. S3 Ruby client: Initial code

```
#!/usr/bin/ruby -w
# S3lib.rb

# Libraries neccessary for making HTTP requests and parsing responses.
require 'rubygems'
require 'rest-open-uri'
```


Spotify USA Inc. Exhibit 1014 Page 48

```
require 'rexml/document'

# Libraries neccessary for request signing
require 'openssl'
require 'digest/sha1'
require 'base64'
require 'uri'

module S3 # This is the beginning of a big, all-encompassing module.

module Authorized
  # Enter your public key (Amazon calls it an "Access Key ID") and
  # your private key (Amazon calls it a "Secret Access Key"). This is
  # so you can sign your S3 requests and Amazon will know who to
  # charge.
  @@public_key = ''
  @@private_key = ''

  if @@public_key.empty? or @@private_key.empty?
    raise "You need to set your S3 keys."
  end

  # You shouldn't need to change this unless you're using an S3 clone like
  # Park Place.
  HOST = 'https://s3.amazonaws.com/'
end
```

The only interesting aspect of this bare-bones `S3::Authorized` is that it's where you should plug in the two cryptographic keys associated with your Amazon Web Services account. Every S3 request you make includes your public key (Amazon calls it an "Access Key ID") so that Amazon can identify you. Every request you make must be cryptographically signed with your private key (Amazon calls it a "Secret Access Key") so that Amazon knows it's really you. I'm using the standard cryptographic terms, even though your "private key" is not totally private—Amazon knows it too. It is private in the sense that you should never reveal it to anyone else. If you do, the person you reveal it to will be able to make S3 requests and have Amazon charge you for it.

The Bucket List

Example 3-4 shows an object-oriented class for my first resource, the list of buckets. I'll call the class for this resource `S3::BucketList`.

Example 3-4. S3 Ruby client: the S3::BucketList class

```
# The bucket list.
class BucketList
  include Authorized

  # Fetch all the buckets this user has defined.
  def get
    buckets = []
```


Spotify USA Inc. Exhibit 1014 Page 49

```
    # GET the bucket list URI and read an XML document from it.
    doc = REXML::Document.new(open(HOST).read)

    # For every bucket...
    REXML::XPath.each(doc, "//Bucket/Name") do |e|
      # ...create a new Bucket object and add it to the list.
      buckets << Bucket.new(e.text) if e.text
    end
    return buckets
  end
end
```

XPath Exposition

Reading from right to left, the XPath expression `//Bucket/Name` means:

Find every Name tag	Name
that's the direct child of a Bucket tag	Bucket/
anywhere in the document.	//

Now my file is a real web service client. If I call `S3::BucketList#get` I make a secure HTTP GET request to *https://s3.amazonaws.com/*, which happens to be the URI of the resource "a list of your buckets." The S3 service sends back an XML document that looks something like Example 3-5. This is a *representation* (as I'll start calling it in the next chapter) of the resource "a list of your buckets." It's just some information about the current state of that list. The `Owner` tag makes it clear whose bucket list it is (my AWS account name is evidently "leonardr28"), and the `Buckets` tag contains a number of `Bucket` tags describing my buckets (in this case, there's one `Bucket` tag and one bucket).

Example 3-5. A sample "list of your buckets"

```
<?xml version='1.0' encoding='UTF-8'?>
<ListAllMyBucketsResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
 <Owner>
  <ID>c0363f7260f2f5fcf38d48039f4fb5cab21b060577817310be5170e7774aad70</ID>
  <DisplayName>leonardr28</DisplayName>
 </Owner>
 <Buckets>
  <Bucket>
   <Name>crummy.com</Name>
   <CreationDate>2006-10-26T18:46:45.000Z</CreationDate>
  </Bucket>
 </Buckets>
</ListAllMyBucketsResult>
```

For purposes of this small client application, the `Name` is the only aspect of a bucket I'm interested in. The XPath expression `//Bucket/Name` gives me the name of every bucket, which is all I need to create `Bucket` objects.

Spotify USA Inc. Exhibit 1014 Page 50

As we'll see, one thing that's missing from this XML document is *links*. The document gives the name of every bucket, but says nothing about where the buckets can be found on the Web. In terms of the REST design criteria, this is the major shortcoming of Amazon S3. Fortunately, it's not too difficult to program a client to calculate a URI from the bucket name. I just follow the rule I gave earlier: `https://s3.amazonaws.com/{name-of-bucket}`.

The Bucket

Now, as shown in Example 3-6, let's write the `S3::Bucket` class, so that `S3::BucketList.get` will have something to instantiate.

Example 3-6. S3 Ruby client: the S3::Bucket class

```
# A bucket that you've stored (or will store) on the S3 application.
class Bucket
  include Authorized
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  # The URI to a bucket is the service root plus the bucket name.
  def uri
    HOST + URI.escape(name)
  end

  # Stores this bucket on S3. Analagous to ActiveRecord::Base#save,
  # which stores an object in the database. See below in the
  # book text for a discussion of acl_policy.
  def put(acl_policy=nil)
    # Set the HTTP method as an argument to open(). Also set the S3
    # access policy for this bucket, if one was provided.
    args = {:method => :put}
    args["x-amz-acl"] = acl_policy if acl_policy

    # Send a PUT request to this bucket's URI.
    open(uri, args)
    return self
  end

  # Deletes this bucket. This will fail with HTTP status code 409
  # ("Conflict") unless the bucket is empty.
  def delete
    # Send a DELETE request to this bucket's URI.
    open(uri, :method => :delete)
  end
```

Here are two more web service methods: `S3::Bucket#put` and `S3::Bucket#delete`. Since the URI to a bucket uniquely identifies the bucket, deletion is simple: you send a DELETE request to the bucket URI, and it's gone. Since a bucket's name goes into its URI,

Spotify USA Inc. Exhibit 1014 Page 51

and a bucket has no other settable properties, it's also easy to create a bucket: just send a PUT request to its URI. As I'll show when I write `S3::Object`, a PUT request is more complicated when not all the data can be stored in the URI.

Earlier I compared my `S3::` classes to ActiveRecord classes, but `S3::Bucket#put` works a little differently from an ActiveRecord implementation of `save`. A row in an ActiveRecord-controlled database table has a numeric unique ID. If you take an ActiveRecord object with ID 23 and change its name, your change is reflected as a change to the database record with ID 23:

```
SET name="newname" WHERE id=23
```

The permanent ID of an S3 bucket is its URI, and the URI includes the name. If you change the name of a bucket and call `put`, the client doesn't rename the old bucket on S3: it creates a new, empty bucket at a new URI with the new name. This is a result of design decisions made by the S3 programmers. It doesn't have to be this way. The Ruby on Rails framework has a different design: when it exposes database rows through a RESTful web service, the URI to a row incorporates its numeric database IDs. If S3 was a Rails service you'd see buckets at URIs like `/buckets/23`. Renaming the bucket wouldn't change the URI.

Now comes the last method of `S3::Bucket`, which I've called `get`. Like `S3::BucketList.get`, this method makes a GET request to the URI of a resource (in this case, a "bucket" resource), fetches an XML document, and parses it into new instances of a Ruby class (see Example 3-7). This method supports a variety of ways to filter the contents of S3 buckets. For instance, you can use `:Prefix` to retrieve only objects whose keys start with a certain string. I won't cover these filtering options in detail. If you're interested in them, see the S3 technical documentation on "Listing Keys."

Example 3-7. S3 Ruby client: the S3::Bucket class (concluded)

```
# Get the objects in this bucket: all of them, or some subset.
#
# If S3 decides not to return the whole bucket/subset, the second
# return value will be set to true. To get the rest of the objects,
# you'll need to manipulate the subset options (not covered in the
# book text).
#
# The subset options are :Prefix, :Marker, :Delimiter, :MaxKeys.
# For details, see the S3 docs on "Listing Keys".
def get(options={})
  # Get the base URI to this bucket, and append any subset options
  # onto the query string.
  uri = uri()
  suffix = '?'

  # For every option the user provided...
  options.each do |param, value|
    # ...if it's one of the S3 subset options...
    if [:Prefix, :Marker, :Delimiter, :MaxKeys].member? :param
      # ...add it to the URI.
      uri << suffix << param.to_s << '=' << URI.escape(value)
```


Spotify USA Inc. Exhibit 1014 Page 52

```
        suffix = '&'
      end
    end

    # Now we've built up our URI. Make a GET request to that URI and
    # read an XML document that lists objects in the bucket.
    doc = REXML::Document.new(open(uri).read)
    there_are_more = REXML::XPath.first(doc, "//IsTruncated").text == "true"

    # Build a list of S3::Object objects.
    objects = []
    # For every object in the bucket...
    REXML::XPath.each(doc, "//Contents/Key") do |e|
      # ...build an S3::Object object and append it to the list.
      objects << Object.new(self, e.text) if e.text
    end
    return objects, there_are_more
  end
end
```

XPath Exposition

Reading from right to left, the XPath expression `//IsTruncated` means:

Find every `IsTruncated` tag	`IsTruncated`
anywhere in the document.	`//`

Make a GET request of the application's root URI, and you get a representation of the resource "a list of your buckets." Make a GET request to the URI of a "bucket" resource, and you get a representation of the bucket: an XML document like the one in Example 3-8, containing a `Contents` tag for every element of the bucket.

Example 3-8. A sample bucket representation

```
<?xml version='1.0' encoding='UTF-8'?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
 <Name>crummy.com</Name>
 <Prefix></Prefix>
 <Marker></Marker>
 <MaxKeys>1000</MaxKeys>
 <IsTruncated>false</IsTruncated>
 <Contents>
  <Key>mydocument</Key>
  <LastModified>2006-10-27T16:01:19.000Z</LastModified>
  <ETag>"93bede57fd3818f93eedce0def329cc7"</ETag>
  <Size>22</Size>
  <Owner>
   <ID>
    c0363f7260f2f5fcf38d48039f4fb5cab21b060577817310be5170e7774aad70</ID>
    <DisplayName>leonardr28</DisplayName>
  </Owner>
```


Spotify USA Inc. Exhibit 1014 Page 53

```
    <StorageClass>STANDARD</StorageClass>
  </Contents>
</ListBucketResult>
```

In this case, the portion of the document I find interesting is the list of a bucket's objects. An object is identified by its key, and I use the XPath expression "//Contents/Key" to fetch that information. I'm also interested in a certain Boolean variable ("//IsTruncated"): whether this document contains keys for every object in the bucket, or whether S3 decided there were too many to send in one document and truncated the list.

Again, the main thing missing from this representation is links. The document lists lots of information about the objects, but not their URIs. The client is expected to know how to turn an object name into that object's URI. Fortunately, it's not too hard to build an object's URI, using the rule I already gave: `https://s3.amazonaws.com/`*`{name-of-bucket}/{name-of-object}`*.

The S3 Object

Now we're ready to implement an interface to the core of the S3 service: the object. Remember that an S3 object is just a data string that's been given a name (a key) and a set of metadata key-value pairs (such as `Content-Type="text/html"`). When you send a GET request to the bucket list, or to a bucket, S3 serves an XML document that you have to parse. When you send a GET request to an object, S3 serves whatever data string you PUT there earlier—byte for byte.

Example 3-9 shows the beginning of `S3::Object`, which should be nothing new by now.

Example 3-9. S3 Ruby client: the S3::Object class

```
# An S3 object, associated with a bucket, containing a value and metadata.
class Object
  include Authorized

  # The client can see which Bucket this Object is in.
  attr_reader :bucket

  # The client can read and write the name of this Object.
  attr_accessor :name

  # The client can write this Object's metadata and value.
  # I'll define the corresponding "read" methods later.
  attr_writer :metadata, :value

  def initialize(bucket, name, value=nil, metadata=nil)
    @bucket, @name, @value, @metadata = bucket, name, value, metadata
  end

  # The URI to an Object is the URI to its Bucket, and then its name.
  def uri
    @bucket.uri + '/' + URI.escape(name)
  end
```


Spotify USA Inc. Exhibit 1014 Page 54

What comes next is my first implementation of an HTTP HEAD request. I use it to fetch an object's metadata key-value pairs and populate the *metadata* hash with it (the actual implementation of `store_metadata` comes at the end of this class). Since I'm using `rest-open-uri`, the code to make the HEAD request looks the same as the code to make any other HTTP request (see Example 3-10).

Example 3-10. S3 Ruby client: the S3::Object#metadata method

```
# Retrieves the metadata hash for this Object, possibly fetching
# it from S3.
def metadata
  # If there's no metadata yet...
  unless @metadata
    # Make a HEAD request to this Object's URI, and read the metadata
    # from the HTTP headers in the response.
    begin
      store_metadata(open(uri, :method => :head).meta)
    rescue OpenURI::HTTPError => e
      if e.io.status == ["404", "Not Found"]
        # If the Object doesn't exist, there's no metadata and this is not
        # an error.
        @metadata = {}
      else
        # Otherwise, this is an error.
        raise e
      end
    end

  end
  return @metadata
end
```

The goal here is to fetch an object's metadata without fetching the object itself. This is the difference between downloading a movie review and downloading the movie, and when you're paying for the bandwidth it's a big difference. This distinction between metadata and representation is not unique to S3, and the solution is general to all resource-oriented web services. The HEAD method gives *any* client a way of fetching the metadata for *any* resource, without also fetching its (possibly enormous) representation.

Of course, sometimes you do want to download the movie, and for that you need a GET request. I've put the GET request in the accessor method `S3::Object#value`, in Example 3-11. Its structure mirrors that of `S3::Object#metadata`.

Example 3-11. S3 Ruby client: the S3::Object#value method

```
# Retrieves the value of this Object, possibly fetching it
# (along with the metadata) from S3.
def value
  # If there's no value yet...
  unless @value
    # Make a GET request to this Object's URI.
    response = open(uri)
```


Spotify USA Inc. Exhibit 1014 Page 55

```
    # Read the metadata from the HTTP headers in the response.
    store_metadata(response.meta) unless @metadata
    # Read the value from the entity-body
    @value = response.read
  end
  return @value
end
```

The client stores objects on the S3 service the same way it stores buckets: by sending a PUT request to a certain URI. The bucket PUT is trivial because a bucket has no distinguishing features other than its name, which goes into the URI of the PUT request. An object PUT is more complex. This is where the HTTP client specifies an object's metadata (such as `Content-Type`) and value. This information will be made available on future HEAD and GET requests.

Fortunately, setting up the PUT request is not terribly complicated, because an object's value is whatever the client says it is. I don't have to wrap the object's value in an XML document or anything. I just send the data as is, and set HTTP headers that correspond to the items of metadata in my `metadata` hash (see Example 3-12).

Example 3-12. S3 Ruby client: the S3::Object#put method

```
# Store this Object on S3.
def put(acl_policy=nil)

  # Start from a copy of the original metadata, or an empty hash if
  # there is no metadata yet.
  args = @metadata ? @metadata.clone : {}

  # Set the HTTP method, the entity-body, and some additional HTTP
  # headers.
  args[:method] = :put
  args["x-amz-acl"] = acl_policy if acl_policy
  if @value
    args["Content-Length"] = @value.size.to_s
    args[:body] = @value
  end

  # Make a PUT request to this Object's URI.
  open(uri, args)
  return self
end
```

The `S3::Object#delete` implementation (see Example 3-13) is identical to `S3::Bucket#delete`.

Example 3-13. S3 Ruby client: the S3::Object#delete method

```
# Deletes this Object.
def delete
  # Make a DELETE request to this Object's URI.
  open(uri, :method => :delete)
end
```


Spotify USA Inc. Exhibit 1014 Page 56

And Example 3-14 shows the method for turning HTTP response headers into S3 object metadata. Except for `Content-Type`, you should prefix all the metadata headers you set with the string "x-amz-meta-". Otherwise they won't make the round trip to the S3 server and back to a web service client. S3 will think they're quirks of your client software and discard them.

Example 3-14. S3 Ruby client: the S3::Object#store_metadata method

```
    private

    # Given a hash of headers from a HTTP response, picks out the
    # headers that are relevant to an S3 Object, and stores them in the
    # instance variable @metadata.
    def store_metadata(new_metadata)
      @metadata = {}
      new_metadata.each do |h,v|
        if RELEVANT_HEADERS.member?(h) || h.index('x-amz-meta') == 0
          @metadata[h] = v
        end
      end
    end
    RELEVANT_HEADERS = ['content-type', 'content-disposition', 'content-range',
                        'x-amz-missing-meta']
  end
```

Request Signing and Access Control

I've put it off as long as I can, and now it's time to deal with S3 authentication. If your main interest is in RESTful services in general, feel free to skip ahead to the section on using the S3 library in clients. But if the inner workings of S3 have piqued your interest, read on.

The code I've shown you so far makes HTTP requests all right, but S3 rejects them, because they don't contain the all-important `Authorization` header. S3 has no proof that you're the owner of your own buckets. Remember, Amazon charges you for the data stored on their servers and the bandwidth used in transferring that data. If S3 accepted requests to your buckets with no authorization, anyone could store data in your buckets and you'd get charged for it.

Most web services that require authentication use a standard HTTP mechanism to make sure you are who you claim to be. But S3's needs are more complicated. With most web services you never want anyone else using your data. But one of the uses of S3 is as a hosting service. You might want to host a big movie file on S3, let anyone download it with their BitTorrent client, and have Amazon send you the bill.

Or you might be *selling* access to movie files stored on S3. Your e-commerce site takes payment from a customer and gives them an S3 URI they can use to download the movie. You're delegating to someone else the right to make a particular web service call (a GET request) as you, and have it charged to your account.

Spotify USA Inc. Exhibit 1014 Page 57

The standard mechanisms for HTTP authentication can't provide security for that kind of application. Normally, the person who's sending the HTTP request needs to know the actual password. You can prevent someone from spying on your password, but you can't say to someone else: "here's my password, but you must promise only to use it to request this one URI."

This is a job for public-key cryptography. Every time you make an S3 request, you use your "private" key (remember, not truly private: Amazon knows it too) to sign the important parts of the request. That'd be the URI, the HTTP method you're using, and a few of the HTTP headers. Only someone with the "private" key can create these signatures for your requests, which is how Amazon knows it's okay to charge you for the request. But once you've signed a request, you can send the signature to a third party without revealing your "private" key. The third party is then free to send an identical HTTP request to the one you signed, and have Amazon charge you for it. In short: someone else can make a specific request as you, for a limited time, without having to know your "private" key.

There is a simpler way to give anonymous access to your S3 objects, and I discuss it below. But there's no way around signing your own requests, so even a simple library like this one must support request signing if it's going to work. I'm reopening the `S3::Authorized` Ruby module now. I'm going to give it the ability to intercept calls to the `open` method, and sign HTTP requests before they're made. Since `S3::BucketList`, `S3::Bucket`, and `S3::Object` have all `included` this module, they'll inherit this ability as soon as I define it. Without the code I'm about to write, all those `open` calls I defined in the classes above will send unsigned HTTP requests that just bounce off S3 with response code 403 ("Forbidden"). With this code, you'll be able to generate signed HTTP requests that pass through S3's security measures (and cost you money). The code in Example 3-15 and the other examples that follow is heavily based on Amazon's own example S3 library.

Example 3-15. S3 Ruby client: the S3::Authorized module

```
module Authorized
  # These are the standard HTTP headers that S3 considers interesting
  # for purposes of request signing.
  INTERESTING_HEADERS = ['content-type', 'content-md5', 'date']

  # This is the prefix for custom metadata headers. All such headers
  # are considered interesting for purposes of request signing.
  AMAZON_HEADER_PREFIX = 'x-amz-'

  # An S3-specific wrapper for rest-open-uri's implementation of
  # open(). This implementation sets some HTTP headers before making
  # the request. Most important of these is the Authorization header,
  # which contains the information Amazon will use to decide who to
  # charge for this request.
  def open(uri, headers_and_options={}, *args, &block)
    headers_and_options = headers_and_options.dup
    headers_and_options['Date'] ||= Time.now.httpdate
    headers_and_options['Content-Type'] ||= ''
```


Spotify USA Inc. Exhibit 1014 Page 58

```
    signed = signature(uri, headers_and_options[:method] || :get,
                       headers_and_options)
    headers_and_options['Authorization'] = "AWS #{@@public_key}:#{signed}"
    Kernel::open(uri, headers_and_options, *args, &block)
  end
```

The tough work here is in the `signature` method, not yet defined. This method needs to construct an encrypted string to go into a request's `Authorization` header: a string that convinces the S3 service that it's really you sending the request—or that you've authorized someone else to make the request at your expense (see Example 3-16).

Example 3-16. S3 Ruby client: the Authorized#signature module

```
# Builds the cryptographic signature for an HTTP request. This is
# the signature (signed with your private key) of a "canonical
# string" containing all interesting information about the request.
def signature(uri, method=:get, headers={}, expires=nil)
  # Accept the URI either as a string, or as a Ruby URI object.
  if uri.respond_to? :path
    path = uri.path
  else
    uri = URI.parse(uri)
    path = uri.path + (uri.query ? "?" + query : "")
  end

  # Build the canonical string, then sign it.
  signed_string = sign(canonical_string(method, path, headers, expires))
end
```

Well, this method passes the buck again, by calling `sign` on the result of `canonical_string`. Let's look at those two methods, starting with `canonical_string`. It turns an HTTP request into a string that looks something like Example 3-17. That string contains everything interesting (from S3's point of view) about an HTTP request, in a specific format. The interesting data is the HTTP method (PUT), the `Content-type` ("text/plain"), a date, a few other HTTP headers ("x-amz-metadata"), and the path portion of the URI ("/crummy.com/myobject"). This is the string that `sign` will sign. Anyone can create this string, but only the S3 account holder and Amazon know how to produce the correct signature.

Example 3-17. The canonical string for a sample request

```
PUT

text/plain
Fri, 27 Oct 2006 21:22:41 GMT
x-amz-metadata:Here's some metadata for the myobject object.
/crummy.com/myobject
```

When Amazon's server receives your HTTP request, it generates the canonical string, signs it (again, Amazon has a copy of your "private" key), and sees whether the two signatures match. That's how S3 authentication works. If the signatures match, your request goes through. Otherwise, you get a response code of 403 ("Forbidden").

Spotify USA Inc. Exhibit 1014 Page 59

Example 3-18 shows the code to generate the canonical string.

Example 3-18. S3 Ruby client: the Authorized#canonical_string method

```
# Turns the elements of an HTTP request into a string that can be
# signed to prove a request comes from your web service account.
def canonical_string(method, path, headers, expires=nil)

  # Start out with default values for all the interesting headers.
  sign_headers = {}
  INTERESTING_HEADERS.each { |header| sign_headers[header] = '' }

  # Copy in any actual values, including values for custom S3
  # headers.
  headers.each do |header, value|
    if header.respond_to? :to_str
      header = header.downcase
      # If it's a custom header, or one Amazon thinks is interesting...
      if INTERESTING_HEADERS.member?(header) ||
          header.index(AMAZON_HEADER_PREFIX) == 0
        # Add it to the header has.
        sign_headers[header] = value.to_s.strip
      end
    end
  end

  # This library eliminates the need for the x-amz-date header that
  # Amazon defines, but someone might set it anyway. If they do,
  # we'll do without HTTP's standard Date header.
  sign_headers['date'] = '' if sign_headers.has_key? 'x-amz-date'

  # If an expiration time was provided, it overrides any Date
  # header. This signature will be valid until the expiration time,
  # not only during the single second designated by the Date header.
  sign_headers['date'] = expires.to_s if expires

  # Now we start building the canonical string for this request. We
  # start with the HTTP method.
  canonical = method.to_s.upcase + "\n"

  # Sort the headers by name, and append them (or just their values)
  # to the string to be signed.
  sign_headers.sort_by { |h| h[0] }.each do |header, value|
    canonical << header << ":" if header.index(AMAZON_HEADER_PREFIX) == 0
    canonical << value << "\n"
  end

  # The final part of the string to be signed is the URI path. We
  # strip off the query string, and (if neccessary) tack one of the
  # special S3 query parameters back on: 'acl', 'torrent', or
  # 'logging'.
  canonical << path.gsub(/\?.*$/, '')

  for param in ['acl', 'torrent', 'logging']
    if path =~ Regexp.new("[&?]#{param}($|&|=)")
      canonical << "?" << param
```


Spotify USA Inc. Exhibit 1014 Page 60

```
        break
      end
    end
    return canonical
  end
```

The implementation of `sign` is just a bit of plumbing around Ruby's standard cryptographic and encoding interfaces (see Example 3-19).

Example 3-19. S3 Ruby client: the Authorized#sign method

```
# Signs a string with the client's secret access key, and encodes the
# resulting binary string into plain ASCII with base64.
def sign(str)
  digest_generator = OpenSSL::Digest::Digest.new('sha1')
  digest = OpenSSL::HMAC.digest(digest_generator, @@private_key, str)
  return Base64.encode64(digest).strip
end
```

Signing a URI

My S3 library has one feature still to be implemented. I've mentioned a few times that S3 lets you sign an HTTP request and give the URI to someone else, letting them make that request as you. Here's the method that lets you do this: `signed_uri` (see Example 3-20). Instead of making an HTTP request with `open`, you pass the `open` arguments into this method, and it gives you a signed URI that anyone can use as you. To limit abuse, a signed URI works only for a limited time. You can customize that time by passing a `Time` object in as the keyword argument `:expires`.

Example 3-20. S3 Ruby client: the Authorized#signed_uri method

```
# Given information about an HTTP request, returns a URI you can
# give to anyone else, to let them them make that particular HTTP
# request as you. The URI will be valid for 15 minutes, or until the
# Time passed in as the :expires option.
def signed_uri(headers_and_options={})
  expires = headers_and_options[:expires] || (Time.now.to_i + (15 * 60))
  expires = expires.to_i if expires.respond_to? :to_i
  headers_and_options.delete(:expires)
  signature = URI.escape(signature(uri, headers_and_options[:method],
                                   headers_and_options, nil))
  q = (uri.index("?")) ? "&" : "?"
  "#{uri}#{q}Signature=#{signature}&Expires=#{expires}&AWSAccessKeyId=#{@@public_key}"
end
end

end # Remember the all-encompassing S3 module? This is the end.
```

Here's how it works. Suppose I want to give a customer access to my hosted file at *https://s3.amazonaws.com/BobProductions/KomodoDragon.avi*. I can run the code in Example 3-21 to generate a URI for my customer.

Spotify USA Inc. Exhibit 1014 Page 61

Example 3-21. Generating a signed URI

```
#!/usr/bin/ruby1.9
# s3-signed-uri.rb
require 'S3lib'

bucket = S3::Bucket.new("BobProductions")
object = S3::Object.new(bucket, "KomodoDragon.avi")
puts object.signed_uri
# "https://s3.amazonaws.com/BobProductions/KomodoDragon.avi
# ?Signature=J%2Fu6kxT3jOzHaFXjsLbowgpzExQ%3D
# &Expires=1162156499&AWSAccessKeyId=0F9DBXKB5274JKTJ8DG2"
```

That URI will be valid for 15 minutes, the default for my `signed_uri` implementation. It incorporates my public key (`AWSAccessKeyId`), the expiration time (`Expires`), and the cryptographic `Signature`. My customer can visit this URI and download the movie file *KomodoDragon.avi*. Amazon will charge me for my customer's use of their bandwidth. If my customer modifies any part of the URI (maybe they to try to download a second movie too), the S3 service will reject their request. An untrustworthy customer can send the URI to all of their friends, but it will stop working in 15 minutes.

You may have noticed a problem here. The canonical string usually includes the value of the `Date` header. When my customer visits the URI you signed, their web browser will surely send a different value for the `Date` header. That's why, when you're generating a canonical string to give to someone else, you set an *expiration* date instead of a request date. Look back to Example 3-18 and the implementation of `canonical_string`, where the expiration date (if provided) overwrites any value for the `Date` header.

Setting Access Policy

What if I want to make an object publicly accessible? I want to serve my files to the world and let Amazon deal with the headaches of server management. Well, I could set an expiration date very far in the future, and give out the enormous signed URI to everyone. But there's an easier way to get the same results: allow anonymous access. You can do this by setting the *access policy* for a bucket or object, telling S3 to respond to unsigned requests for it. You do this by sending the `x-amz-acl` header along with the PUT request that creates the bucket or object.

That's what the `acl_policy` argument to `Bucket#put` and `Object#put` does. If you want to make a bucket or object publicly readable or writable, you pass an appropriate value in for `acl_policy`. My client sends that value as part of the custom HTTP request header `X-amz-acl`. Amazon S3 reads this request header and sets the rules for bucket or object access appropriately.

The client in Example 3-22 creates an S3 object that anyone can read by visiting its URI at `https://s3.amazonaws.com/BobProductions/KomodoDragon-Trailer.avi`. In this scenario, I'm not selling my movies: just using Amazon as a hosting service so I don't have to serve movies from my own web site.

Spotify USA Inc. Exhibit 1014 Page 62

Example 3-22. Creating a publicly-readable object

```
#!/usr/bin/ruby -w
# s3-public-object.rb
require 'S3lib'

bucket = S3::Bucket.new("BobProductions")
object = S3::Object.new(bucket, "KomodoDragon-Trailer.avi")
object.put("public-read")
```

S3 understands four access policies:

`private`
: The default. Only requests signed by your "private" key are accepted.

`public-read`
: Unsigned GET requests are accepted: anyone can download an object or list a bucket.

`public-write`
: Unsigned GET and PUT requests are accepted. Anyone can modify an object, or add objects to a bucket.

`authenticated-read`
: Unsigned requests are rejected, but read requests can be signed by the "private" key of *any* S3 user, not just your own. Basically, anyone with an S3 account can download your object or list your bucket.

There are also fine-grained ways of granting access to a bucket or object, which I won't cover. If you're interested, see the section "Setting Access Policy with REST" in the S3 technical documentation. That section reveals a parallel universe of extra resources. Every bucket `/{name-of-bucket}` has a shadow resource `/{name-of-bucket}?acl` corresponding to that bucket's access control rules, and every object `/{name-of-bucket}/{name-of-object}` has a shadow ACL resource `/{name-of-bucket}/{name-of-object}?acl`. By sending PUT requests to these URIs, and including XML representations of access control lists in the request entity-bodies, you can set specific permissions and limit access to particular S3 users.

Using the S3 Client Library

I've now shown you a Ruby client library that can access just about the full capabilities of Amazon's S3 service. Of course, a library is useless without clients that use it. In the previous section I showed you a couple of small clients to demonstrate points about security, but now I'd like to show something a little more substantial.

Example 3-23 is a simple command-line S3 client that can create a bucket and an object, then list the contents of the bucket. This client should give you a high-level picture of how S3's resources work together. I've annotated the lines of code that trigger HTTP requests, by describing the HTTP requests in comments off to the right.

Spotify USA Inc. Exhibit 1014 Page 63

Example 3-23. A sample S3 client

```
#!/usr/bin/ruby -w
# s3-sample-client.rb
require 'S3lib'

# Gather command-line arguments
bucket_name, object_name, object_value = ARGV
unless bucket_name
  puts "Usage: #{$0} [bucket name] [object name] [object value]"
  exit
end

# Find or create the bucket.
buckets = S3::BucketList.new.get                # GET /
bucket = buckets.detect { |b| b.name == bucket_name }
if bucket
  puts "Found bucket #{bucket_name}."
else
  puts "Could not find bucket #{bucket_name}, creating it."
  bucket = S3::Bucket.new(bucket_name)
  bucket.put                                    # PUT /{bucket}
end

# Create the object.
object = S3::Object.new(bucket, object_name)
object.metadata['content-type'] = 'text/plain'
object.value = object_value
object.put                                      # PUT /{bucket}/{object}

# For each object in the bucket...
bucket.get[0].each do |o|                       # GET /{bucket}
  # ...print out information about the object.
  puts "Name: #{o.name}"
  puts "Value: #{o.value}"                      # GET /{bucket}/{object}
  puts "Metadata hash: #{o.metadata.inspect}"
  puts
end
```

Clients Made Transparent with ActiveResource

Since all RESTful web services expose basically the same simple interface, it's not a big chore to write a custom client for every web service. It is a little wasteful, though, and there are two alternatives. You can describe a service with a WADL file (introduced in the previous chapter, and covered in more detail in Chapter 9), and then access it with a generic WADL client. There's also a Ruby library called ActiveResource that makes it trivial to write clients for certain kinds of web services.

ActiveResource is designed to run against web services that expose the rows and tables of a relational database. WADL can describe almost any kind of web service, but ActiveResource only works as a client for web services that follow certain conventions. Right now, Ruby on Rails is the only framework that follows the conventions. But any

Spotify USA Inc. Exhibit 1014 Page 64

web service can answer requests from an ActiveResource client: it just has to expose its database through the same RESTful interface as Rails.

As of the time of writing, there are few publicly available web services that can be used with an ActiveResource client (I list a couple in Appendix A). To show you an example I'm going create a small Rails web service of my own. I'll be able to drive my service with an ActiveResource client, without writing any HTTP client or XML parsing code.

Creating a Simple Service

My web service will be a simple notebook: a way of keeping timestamped notes to myself. I've got Rails 1.2 installed on my computer, so I can create the notebook service like this:

```
$ rails notebook
$ cd notebook
```

I create a database on my system called `notebook_development`, and edit the Rails file *notebook/config/database.yml* to give Rails the information it needs to connect to my database. Any general guide to Rails will have more detail on these initial steps.

Now I've created a Rails application, but it doesn't do anything. I'm going to generate code for a simple, RESTful web service with the `scaffold_resource` generator. I want my notes to contain a timestamp and a body of text, so I run the following command:

```
$ ruby script/generate scaffold_resource note date:date body:text
create  app/views/notes
create  app/views/notes/index.rhtml
create  app/views/notes/show.rhtml
create  app/views/notes/new.rhtml
create  app/views/notes/edit.rhtml
create  app/views/layouts/notes.rhtml
create  public/stylesheets/scaffold.css
create  app/models/note.rb
create  app/controllers/notes_controller.rb
create  test/functional/notes_controller_test.rb
create  app/helpers/notes_helper.rb
create  test/unit/note_test.rb
create  test/fixtures/notes.yml
create  db/migrate
create  db/migrate/001_create_notes.rb
route  map.resources :notes
```

Rails has generated a complete set of web service code—model, view, and controller—for my "note" object. There's code in `db/migrate/001_create_notes.rb` that creates a database table called `notes` with three fields: a unique ID, a date (`date`), and a piece of text (`body`).

The model code in `app/models/note.rb` provides an ActiveResource interface to the database table. The controller code in `app/controllers/notes_controller.rb` exposes that interface to the world through HTTP, and the views in *app/views/notes* define the

Spotify USA Inc. Exhibit 1014 Page 65

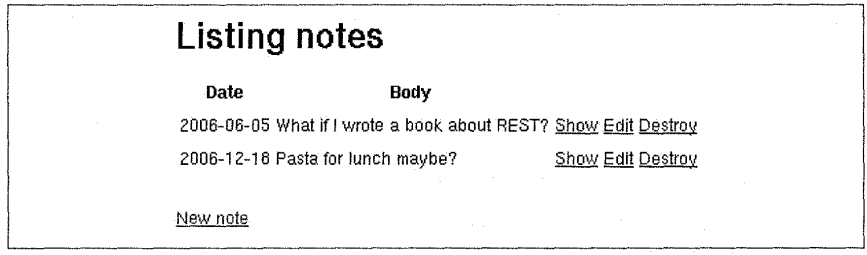


Figure 3-1. The notebook web application with a few entered notes

user interface. It adds up to a RESTful web service—not a very fancy one, but one that's good enough for a demo or to use as a starting point.

Before starting the service I need to initialize the database:

```
$ rake db:migrate
== CreateNotes: migrating =====================================================
-- create_table(:notes)
   -> 0.0119s
== CreateNotes: migrated (0.0142s) ============================================
```

Now I can start the notebook application and start using my service:

```
$ script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
```

An ActiveResource Client

The application I just generated is not much use except as a demo, but it demos some pretty impressive features. First, it's both a web service and a web application. I can visit *http://localhost:3000/notes* in my web browser and create notes through the web interface. After a while the view of *http://localhost:3000/notes* might look like Figure 3-1.

If you've ever written a Rails application or seen a Rails demo, this should look familiar. But in Rails 1.2, the generated model and controller can also act as a RESTful web service. A programmed client can access it as easily as a web browser can.

Unfortunately, the ActiveResource client itself was not released along with Rails 1.2. As of the time of writing, it's still being developed on the tip of the Rails development tree. To get the code I need to check it out from the Subversion version control repository:

```
$ svn co http://dev.rubyonrails.org/svn/rails/trunk activeresource_client
$ cd activeresource_client
```


Spotify USA Inc. Exhibit 1014 Page 66

Now I'm ready to write ActiveResource clients for the notebook's web service. Example 3-24 is a client that creates a note, modifies it, lists the existing notes, and then deletes the note it just created.

Example 3-24. An ActiveResource client for the notebook service

```
#!/usr/bin/ruby -w
# activeresource-notebook-manipulation.rb

require 'activesupport/lib/active_support'
require 'activeresource/lib/active_resource'

# Define a model for the objects exposed by the site
class Note < ActiveResource::Base
  self.site = 'http://localhost:3000/'
end

def show_notes
  notes = Note.find :all                # GET /notes.xml
  puts "I see #{notes.size} note(s):"
  notes.each do |note|
    puts " #{note.date}: #{note.body}"
  end
end

new_note = Note.new(:date => Time.now, :body => "A test note")
new_note.save                           # POST /notes.xml

new_note.body = "This note has been modified."
new_note.save                           # PUT /notes/{id}.xml

show_notes

new_note.destroy                        # DELETE /notes/{id}.xml

puts
show_notes
```

Example 3-25 shows the output when I run that program:

Example 3-25. A run of activeresource-notebook-manipulation.rb

```
I see 3 note(s):
 2006-06-05: What if I wrote a book about REST?
 2006-12-18: Pasta for lunch maybe?
 2006-12-18: This note has been modified.

I see 2 note(s):
 2006-06-05: What if I wrote a book about REST?
 2006-12-18: Pasta for lunch maybe?
```

If you're familiar with ActiveRecord, the object-relational mapper that connects Rails to a database, you'll notice that the ActiveResource interface looks almost exactly the same. Both libraries provide an object-oriented interface to a wide variety of objects, each of which exposes a uniform interface. With ActiveRecord, the objects live in a

Spotify USA Inc. Exhibit 1014 Page 67

database and are exposed through SQL, with its SELECT, INSERT, UPDATE, and DELETE. With ActiveResource, they live in a Rails application and are exposed through HTTP, with its GET, POST, PUT, and DELETE.

Example 3-26 is an excerpt from the Rails server logs at the time I ran my ActiveResource client. The GET, POST, PUT, and DELETE requests correspond to the commented lines of code back in Example 3-24.

Example 3-26. The HTTP requests made by activeresource-notebook-manipulation.rb

```
"POST /notes.xml HTTP/1.1" 201
"PUT /notes/5.xml HTTP/1.1" 200
"GET /notes.xml HTTP/1.1" 200
"DELETE /notes/5.xml HTTP/1.1" 200
"GET /notes.xml HTTP/1.1" 200
```

What's going on in these requests? The same thing that's going on in requests to S3: resource access through HTTP's uniform interface. My notebook service exposes two kinds of resources:

- The list of notes (`/notes.xml`). Compare to an S3 bucket, which is a list of objects.
- A note (`/notes/{id}.xml`). Compare to an S3 object.

These resources expose GET, PUT, and DELETE, just like the S3 resources do. The list of notes also supports POST to create a new note. That's a little different from S3, where objects are created with PUT, but it's just as RESTful.

When the client runs, XML documents are transferred invisibly between client and server. They look like the documents in Example 3-27 or 3-28: simple depictions of the underlying database rows.

Example 3-27. The response entity-body from a GET request to /notes.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<notes>
 <note>
  <body>What if I wrote a book about REST?</body>
  <date type="date">2006-06-05</date>
  <id type="integer">2</id>
 </note>
 <note>
  <body>Pasta for lunch maybe?</body>
  <date type="date">2006-12-18</date>
  <id type="integer">3</id>
 </note>
</notes>
```

Example 3-28. A request entity-body sent as part of a PUT request to /notes/5.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
 <body>This note has been modified.</body>
</note>
```


Spotify USA Inc. Exhibit 1014 Page 68

A Python Client for the Simple Service

Right now the only ActiveResource client library is the Ruby library, and Rails is the only framework that exposes ActiveResource-compatible services. But nothing's happening here except HTTP requests that pass XML documents into certain URIs and get XML documents back. There's no reason why a client in some other language couldn't send those XML documents, or why some other framework couldn't expose the same URIs.

Example 3-29 is a Python implementation of the client program from Example 3-24. It's longer than the Ruby program, because it can't rely on ActiveResource. It has to build its own XML documents and make its own HTTP requests, but its structure is almost exactly the same.

Example 3-29. A Python client for an ActiveResource service

```
#!/usr/bin/python
# activeresource-notebook-manipulation.py

from elementtree.ElementTree import Element, SubElement, tostring
from elementtree import ElementTree
import httplib2
import time

BASE = "http://localhost:3000/"
client = httplib2.Http(".cache")

def showNotes():
    headers, xml = client.request(BASE + "notes.xml")
    doc = ElementTree.fromstring(xml)
    for note in doc.findall('note'):
        print "%s: %s" % (note.find('date').text, note.find('body').text)

newNote = Element("note")
date = SubElement(newNote, "date")
date.attrib['type'] = "date"
date.text = time.strftime("%Y-%m-%d", time.localtime())
body = SubElement(newNote, "body")
body.text = "A test note"

headers, ignore = client.request(BASE + "notes.xml", "POST",
                                 body= tostring(newNote),
                                 headers={'content-type' : 'application/xml'})
newURI = headers['location']

modifiedBody = Element("note")
body = SubElement(modifiedBody, "body")
body.text = "This note has been modified"

client.request(newURI, "PUT",
               body=tostring(modifiedBody),
               headers={'content-type' : 'application/xml'})

showNotes()
```


Spotify USA Inc. Exhibit 1014 Page 69

```
client.request(newURI, "DELETE")

print
showNotes()
```

Parting Words

Because RESTful web services have simple and well-defined interfaces, it's not difficult to clone them or swap out one implementation for another. Park Place (*http://code.whytheluckystiff.net/parkplace*) is a Ruby application that exposes the same HTTP interface as S3. You can use Park Place to host your own version of S3. S3 libraries and client programs will work against your Park Place server just as they now do against *https://s3.amazonaws.com/*.

It's also possible to clone ActiveResource. No one has done this yet, but it shouldn't be difficult to write a general ActiveResource client for Python or any other dynamic language. In the meantime, writing a one-off client for an ActiveResource-compatible service is no more difficult than writing a client for any other RESTful service.

By now you should feel comfortable with the prospect of writing a client for any RESTful or REST-RPC hybrid service, whether it serves XML, HTML, JSON, or some mixture. It's all just HTTP requests and document parsing.

You should also be getting a feel for what differentiates RESTful web services like S3 and Yahoo!'s search services from RPC-style and hybrid services like the Flickr and del.icio.us APIs. This is not a judgement about the service's content, only about its architecture. In woodworking it's important to work with the grain of the wood. The Web, too, has a grain, and a RESTful web service is one that works with it.

In the coming chapters I'll show how you can create web services that are more like S3 and less like the del.icio.us API. This culminates in Chapter 7, which reinvents del.icio.us as a RESTful web service.

Spotify USA Inc. Exhibit 1014 Page 70

CHAPTER 4

The Resource-Oriented Architecture

I've shown you the power of REST, but I haven't shown you in any systematic way how that power is structured or how to expose it. In this chapter I outline a concrete RESTful architecture: the Resource-Oriented Architecture (ROA). The ROA is a way of turning a problem into a RESTful web service: an arrangement of URIs, HTTP, and XML that works like the rest of the Web, and that programmers will enjoy using.

In Chapter 1 I classified RESTful web services by their answers to two questions. These answers correspond to two of the four defining features of REST:

- The scoping information ("why should the server send this data instead of that data?") is kept in the URI. This is the principle of *addressability*.
- The method information ("why should the server send this data instead of deleting it?") is kept in the HTTP method. There are only a few HTTP methods, and everyone knows ahead of time what they do. This is the principle of the *uniform interface*.

In this chapter I introduce the moving parts of the Resource-Oriented Architecture: resources (of course), their names, their representations, and the links between them. I explain and promote the properties of the ROA: addressability, statelessness, connectedness, and the uniform interface. I show how the web technologies (HTTP, URIs, and XML) implement the moving parts to make the properties possible.

In the previous chapters I illustrated concepts by pointing to existing web services, like S3. I continue that tradition in this chapter, but I'll also illustrate concepts by pointing to existing web sites. Hopefully I've convinced you by now that web sites are web services, and that many web applications (such as search engines) are RESTful web services. When I talk about abstract concepts like addressability, it's useful to show you real URIs, which you can type into your web browser to see the concepts in action.

Resource-Oriented What Now?

Why come up with a new term, Resource-Oriented Architecture? Why not just say REST? Well, I do say REST, on the cover of this book, and I hold that everything in the

Spotify USA Inc. Exhibit 1014 Page 71

Resource-Oriented Architecture is also RESTful. But REST is not an architecture: it's a set of design criteria. You can say that one architecture meets those criteria better than another, but there is no one "REST architecture."

Up to now, people have tended to mint one-off architectures as they design their services, according to their own understandings of REST. The most obvious outcome of this is the wide variety of REST-RPC hybrid web services that their creators claim are RESTful. I'm trying to put a stop to that by presenting a set of concrete rules for building web services that really will be RESTful. In the next two chapters I'll even show simple procedures you can follow to turn requirements into resources. If you don't like my rules, you'll at least have an idea of what you can change and stay RESTful.

As a set of design criteria, REST is very general. In particular, it's not tied to the Web. Nothing about REST depends on the mechanics of HTTP or the structure of URIs. But I'm talking about *web* services, so I explicitly tie the Resource-Oriented Architecture to the technologies of the Web. I want to talk about how to do REST with HTTP and URIs, in specific programming languages. If the future produces RESTful architectures that don't run on top of the Web, their best practices will probably look similar to the ROA, but the details will be different. We'll cross that bridge when we come to it.

The traditional definition of REST leaves a lot of open space, which practitioners have seeded with folklore. I deliberately go further than Roy Fielding in his dissertation, or the W3C in their standards: I want to clear some of that open space so that the folklore has room to grow into a well-defined set of best practices. Even if REST were an architecture, it wouldn't be fair to call my architecture by the same name. I'd be tying my empirical observations and suggestions to the more general thoughts of those who built the Web.

My final reason for coming up with a new term is that "REST" is a term used in religious nerd wars. When it's used, the implication is usually that there is one true RESTful architecture and it's the one the speaker prefers. People who prefer another RESTful architecture disagree. The REST community fragments, despite a general agreement on basic things like the value of URIs and HTTP.

Ideally there would be no religious wars, but I've seen enough to know that wishing won't end them. So I'm giving a distinctive name to my philosophy of how RESTful applications should be designed. When these ideas are, inevitably, used as fodder in wars, people who disagree with me can address aspects of the Resource-Oriented Architecture separate from other RESTful architectures, and from REST in general. Clarity is the first step toward understanding.

The phrases "resource-oriented" and "resource-oriented architecture" have been used to describe RESTful architectures in general.* I don't claim that "Resource-Oriented Architecture" is a completely original term, but I think that my usage meshes well with preexisting uses, and that it's better to use this term than claim to speak for REST as a whole.

Spotify USA Inc. Exhibit 1014 Page 72

What's a Resource?

A resource is anything that's important enough to be referenced as a thing in itself. If your users might "want to create a hypertext link to it, make or refute assertions about it, retrieve or cache a representation of it, include all or part of it by reference into another representation, annotate it, or perform other operations on it", then you should make it a resource.†

Usually, a resource is something that can be stored on a computer and represented as a stream of bits: a document, a row in a database, or the result of running an algorithm. A resource may be a physical object like an apple, or an abstract concept like courage, but (as we'll see later) the representations of such resources are bound to be disappointing.

Here are some possible resources:

- Version 1.0.3 of the software release
- The latest version of the software release
- The first weblog entry for October 24, 2006
- A road map of Little Rock, Arkansas
- Some information about jellyfish
- A directory of resources pertaining to jellyfish
- The next prime number after 1024
- The next five prime numbers after 1024
- The sales numbers for Q42004
- The relationship between two acquaintances, Alice and Bob
- A list of the open bugs in the bug database

URIs

What makes a resource a resource? *It has to have at least one URI.* The URI is the name and address of a resource. If a piece of information doesn't have a URI, it's not a resource and it's not really on the Web, except as a bit of data describing some other resource.

* The earliest instance of "resource-oriented" I've found is a 2004 IBM developerWorks article by James Snell: "Resource-oriented vs. activity-oriented Web services" (*http://www-128.ibm.com/developerworks/xml/library/ws-restvsoap/*). Alex Bunardzic used "Resource-Oriented Architecture" in August 2006, before this book was announced: *http://jooto.com/blog/index.php/2006/08/08/replacing-service-oriented-architecture-with-resource-oriented-architecture/*. I don't agree with everything in those articles, but I do acknowledge their priority in terminology.

† "The Architecture of the World Wide Web" (*http://www.w3.org/TR/2004/REC-webarch-20041215/#p39*), which is full of good quotes, incidentally: "Software developers should expect that sharing URIs across applications will be useful, even if that utility is not initially evident." This could be the battle cry of the ROA.

Spotify USA Inc. Exhibit 1014 Page 73

Remember the sample session in Preface, when I was making fun of HTTP 0.9? Let's say this is a HTTP 0.9 request for `http://www.example.com/hello.txt`:

Client request	Server response
`GET /hello.txt`	`Hello, world!`

An HTTP client manipulates a resource by connecting to the server that hosts it (in this case, `www.example.com`), and sending the server a method ("GET") and a path to the resource ("/hello.txt"). Today's HTTP 1.1 is a little more complex than 0.9, but it works the same way. Both the server and the path come from the resource's URI.

Client request	Server response
`GET /hello.txt HTTP/1.1` `Host: www.example.com`	`200 OK` `Content-Type: text/plain` `Hello, world!`

The principles behind URIs are well described by Tim Berners-Lee in Universal Resource Identifiers—Axioms of Web Architecture (*http://www.w3.org/DesignIssues/Axioms*). In this section I expound the principles behind constructing URIs and assigning them to resources.



The URI is the fundamental technology of the Web. There were hypertext systems before HTML, and Internet protocols before HTTP, but they didn't talk to each other. The URI interconnected all these Internet protocols into a Web, the way TCP/IP interconnected networks like Usenet, Bitnet, and CompuServe into a single Internet. Then the Web co-opted those other protocols and killed them off, just like the Internet did with private networks.

Today we surf the Web (not Gopher), download files from the Web (not FTP sites), search publications from the Web (not WAIS), and have conversations on the Web (not Usenet newsgroups). Version control systems like Subversion and arch work over the Web, as opposed to the custom CVS protocol. Even email is slowly moving onto the Web.

The web kills off other protocols because it has something most protocols lack: a simple way of labeling every available item. Every resource on the Web has at least one URI. You can stick a URI on a billboard. People can see that billboard, type that URI into their web browsers, and go right to the resource you wanted to show them. It may seem strange, but this everyday interaction was impossible before URIs were invented.

Spotify USA Inc. Exhibit 1014 Page 74

URIs Should Be Descriptive

Here's the first point where the ROA builds upon the sparse recommendations of the REST thesis and the W3C recommendations. I propose that a resource and its URI ought to have an intuitive correspondence. Here are some good URIs for the resources I listed above:

- `http://www.example.com/software/releases/1.0.3.tar.gz`
- `http://www.example.com/software/releases/latest.tar.gz`
- `http://www.example.com/weblog/2006/10/24/0`
- `http://www.example.com/map/roads/USA/AR/Little_Rock`
- `http://www.example.com/wiki/Jellyfish`
- `http://www.example.com/search/Jellyfish`
- `http://www.example.com/nextprime/1024`
- `http://www.example.com/next-5-primes/1024`
- `http://www.example.com/sales/2004/Q4`
- `http://www.example.com/relationships/Alice;Bob`
- `http://www.example.com/bugs/by-state/open`

URIs should have a structure. They should vary in predictable ways: you should not go to `/search/Jellyfish` for jellyfish and `/i-want-to-know-about/Mice` for mice. If a client knows the structure of the service's URIs, it can create its own entry points into the service. This makes it easy for clients to use your service in ways you didn't think of.

This is not an absolute rule of REST, as we'll see in the "Name the Resources" section. URIs do not technically have to have any structure or predictability, but I think they should. This is one of the rules of good web design, and it shows up in RESTful and REST-RPC hybrid services alike.

The Relationship Between URIs and Resources

Let's consider some edge cases. Can two resources be the same? Can two URIs designate the same resource? Can a single URI designate two resources?

By definition, no two resources can be the same. If they were the same, you'd only have one resource. However, at some moment in time two different resources may point to the same data. If the current software release is 1.0.3, then *http://www.example.com/software/releases/1.0.3.tar.gz* and *http://www.example.com/software/releases/latest.tar.gz* will refer to the same file for a while. But the ideas behind those two URIs are different: one of them always points to a particular version, and the other points to whatever version is newest at the time the client accesses it. That's two concepts and two resources. You wouldn't link to `latest` when reporting a bug in version 1.0.3.

Spotify USA Inc. Exhibit 1014 Page 75

A resource may have one URI or many. The sales numbers available at *http://www.example.com/sales/2004/Q4* might also be available at *http://www.example.com/sales/Q42004*. If a resource has multiple URIs, it's easier for clients to refer to the resource. The downside is that each additional URI dilutes the value of all the others. Some clients use one URI, some use another, and there's no automatic way to verify that all the URIs refer to the same resource.



One way to get around this is to expose multiple URIs for the same resource, but have one of them be the "canonical" URI for that resource. When a client requests the canonical URI, the server sends the appropriate data along with response code of 200 ("OK"). When a client requests one of the other URIs, the server sends a response code 303 ("See Also") along with the canonical URI. The client can't see whether two URIs point to the same resource, but it can make two HEAD requests and see if one URI redirects to the other or if they both redirect to a third URI.

Another way is to serve all the URIs as though they were the same, but give the "canonical" URI in the `Content-Location` response header whenever someone requests a non-canonical URI.

Fetching `sales/2004/Q4` might get you the same bytestream as fetching `sales/Q42004`, because they're different URIs for the same resource: "sales for the last quarter of 2004." Fetching `releases/1.0.3.tar.gz` might give you the exact same bytestream as fetching `releases/latest.tar.gz`, but they're different resources because they represent different things: "version 1.0.3" and "the latest version."

Every URI designates exactly one resource. If it designated more than one, it wouldn't be a *Universal* Resource Identifier. However, when you fetch a URI the server may send you information about multiple resources: the one you requested and other, related ones. When you fetch a web page, it usually conveys some information of its own, but it also has links to other web pages. When you retrieve an S3 bucket with an Amazon S3 client, you get a document that contains information about the bucket, and information about related resources: the objects in the bucket.

Addressability

Now that I've introduced resources and their URIs, I can go in depth into two of the features of the ROA: addressability and statelessness.

An application is addressable if it exposes the interesting aspects of its data set as resources. Since resources are exposed through URIs, an addressable application exposes a URI for every piece of information it might conceivably serve. This is usually an infinite number of URIs.

Spotify USA Inc. Exhibit 1014 Page 76

From the end-user perspective, addressability is the most important aspect of any web site or application. Users are clever, and they'll overlook or work around almost any deficiency if the data is interesting enough, but it's very difficult to work around a lack of addressability.

Consider a real URI that names a resource in the genre "directory of resources about jellyfish": *http://www.google.com/search?q=jellyfish*. That jellyfish search is just as much a real URI as *http://www.google.com*. If HTTP wasn't addressable, or if the Google search engine wasn't an addressable web application, I wouldn't be able to publish that URI in a book. I'd have to tell you: "Open a web connection to `google.com`, type 'jellyfish' in the search box, and click the 'Google Search' button."



This isn't an academic worry. Until the mid-1990s, when `ftp://` URIs became popular for describing files on FTP sites, people had to write things like: "Start an anonymous FTP session on `ftp.example.com`. Then change to directory `pub/files/` and download file *file.txt*." URIs made FTP as addressable as HTTP. Now people just write: "Download *ftp://ftp.example.com/pub/files/file.txt*." The steps are the same, but now they can be carried out by machine.

But HTTP and Google are both addressable, so I can print that URI in a book. You can read it and type it in. When you do, you end up where I was when I went through the Google web application.

You can then bookmark that page and come back to it later. You can link to it on a web page of your own. You can email the URI to someone else. If HTTP wasn't addressable, you'd have to download the whole page and send the HTML file as an attachment.

To save bandwidth, you can set up an HTTP proxy cache on your local network. The first time someone requests *http://www.google.com/search?q=jellyfish*, the cache will save a local copy of the document. The next time someone hits that URI, the cache might serve the saved copy instead of downloading it again. These things are possible only if every page has a unique identifying string: an address.

It's even possible to chain URIs: to use one URI as input to another one. You can use an external web service to validate a page's HTML, or to translate its text into another language. These web services expect a URI as input. If HTTP wasn't addressable, you'd have no way of telling them which resource you wanted them to operate on.

Amazon's S3 service is addressable because every bucket and every object has its own URI, as does the bucket list. Buckets and objects that don't exist yet aren't yet resources, but they too have their own URIs: you can create a resource by sending a PUT request to its URI.

The filesystem on your home computer is another addressable system. Command-line applications can take a path to a file and do strange things to it. The cells in a spreadsheet

Spotify USA Inc. Exhibit 1014 Page 77

are also addressable; you can plug the name of a cell into a formula, and the formula will use whatever value is currently in that cell. URIs are the file paths and cell addresses of the Web.

Addressability is one of the best things about web applications. It makes it easy for clients to use web sites in ways the original designers never imagined. Following this one rule gives you and your users many of the benefits of REST. This is why REST-RPC services are so common: they combine addressability with the procedure-call programming model. It's why I gave resources top billing in the name of the Resource-Oriented Architecture: because resources are the kind of thing that's addressable.

This seems natural, the way the Web should work. Unfortunately, many web applications *don't* work this way. This is especially true of Ajax applications. As I show in Chapter 11, most Ajax applications are just clients for RESTful or hybrid web services. But when you use these clients as though they are web sites, you notice that they don't *feel* like web sites.

No need to pick on the little guys; let's continue our tour of the Google properties by considering the Gmail online email service. From the end-user perspective, there is only one Gmail URI: *https://mail.google.com/*. Whatever you do, whatever pieces of information you retrieve from or upload to Gmail, you'll never see a different URI. The resource "email messages about jellyfish" isn't addressable, the way Google's "web pages about jellyfish" is.‡ Yet behind the scenes, as I show in Chapter 11, is a web site that is addressable. The list of email messages about jellyfish *does* have a URI: it's *https://mail.google.com/mail/?q=jellyfish&search=query&view=tl*. The problem is, you're not the consumer of that web site. The web site is really a web service, and the real consumer is a JavaScript program running inside your web browser.§ The Gmail web service is addressable, but the Gmail web application that uses it is not.

Statelessness

Addressability is one of the four main features of the ROA. The second is statelessness. I'll give you two definitions of statelessness: a somewhat general definition and a more practical definition geared toward the ROA.

Statelessness means that every HTTP request happens in complete isolation. When the client makes an HTTP request, it includes all information neccessary for the server to fulfill that request. The server never relies on information from previous requests. If that information was important, the client would have sent it again in this request.

‡ Compare the Ajax interface against the more addressable version of Gmail you get by starting off at the URI *https://mail.google.com/mail/?ui=html*. If you use this plain HTML interface, the resource "email messages about jellyfish" *is* addressable.

§ Other consumers of this web service include the libgmail library for Python (*http://libgmail.sourceforge.net/*).

Spotify USA Inc. Exhibit 1014 Page 78

More practically, consider statelessness in terms of addressability. Addressability says that every interesting piece of information the server can provide should be exposed as a resource, and given its own URI. Statelessness says that the *possible states* of the server are also resources, and should be given their own URIs. The client should not have to coax the server into a certain state to make it receptive to a certain request.

On the human web, you often run into situations where your browser's back button doesn't work correctly, and you can't go back and forth in your browser history. Sometimes this is because you performed an irrevocable action, like posting a weblog entry or buying a book, but often it's because you're at a web site that violates the principle of statelessness. Such a site expects you to make requests in a certain order: A, B, then C. It gets confused when you make request B a second time instead of moving on to request C.

Let's take the search example again. A search engine is a web service with an infinite number of possible states: at least one for every string you might search for. Each state has its own URI. You can ask the service for a directory of resources about mice: *http://www.google.com/search?q=mice*. You can ask for a directory of resources about jellyfish: *http://www.google.com/search?q=jellyfish*. If you're not comfortable creating a URI from scratch, you can ask the service for a form to fill out: *http://www.google.com/*.

When you ask for a directory of resources about mice or jellyfish, you don't get the whole directory. You get a single *page* of the directory: a list of the 10 or so items the search engine considers the best matches for your query. To get more of the directory you must make more HTTP requests. The second and subsequent pages are distinct states of the application, and they need to have their own URIs: something like *http://www.google.com/search?q=jellyfish&start=10*. As with any addressable resource, you can transmit that state of the application to someone else, cache it, or bookmark it and come back to it later.

Figure 4-1 is a simple state diagram showing how an HTTP client might interact with four states of a search engine.

This is a stateless application because every time the client makes a request, it ends up back where it started. Each request is totally disconnected from the others. The client can make requests for these resources any number of times, in any order. It can request page 2 of "mice" before requesting page 1 (or not request page 1 at all), and the server won't care.

By way of contrast, Figure 4-2 shows the same states arranged statefully, with states leading sensibly into each other. Most desktop applications are designed this way.

That's a lot better organized, and if HTTP were designed to allow stateful interaction, HTTP requests could be a lot simpler. When the client started a session with the search engine it could be automatically fed the search form. It wouldn't have to send any request data at all, because the first response would be predetermined. If the client was looking at the first 10 entries in the mice directory and wanted to see entries 11–20, it

Spotify USA Inc. Exhibit 1014 Page 79

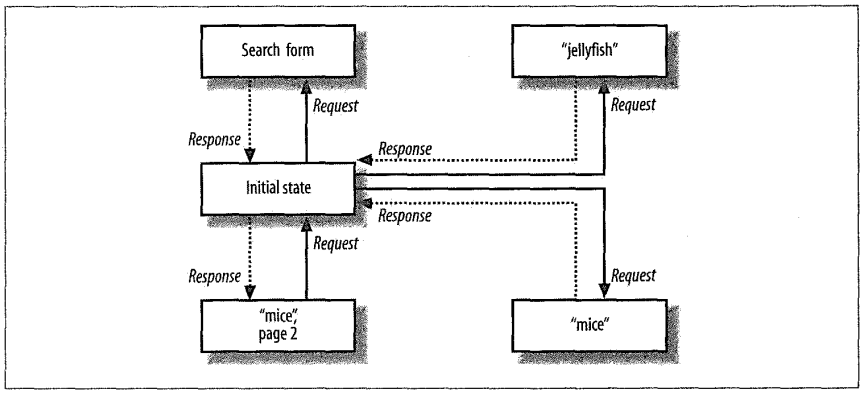


Figure 4-1. A stateless search engine

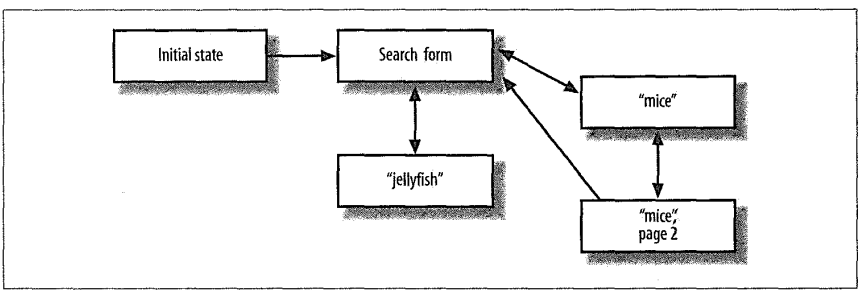


Figure 4-2. A stateful search engine

could just send a request that said "start=10". It wouldn't have to send `/search?q=mice&start=10`, repeating the intitial assertions: "I'm searching, and searching for mice in particular."

FTP works this way: it has a notion of a "working directory" that stays constant over the course of a session unless you change it. You might log in to an FTP server, `cd` to a certain directory, and `get` a file from that directory. You can `get` another file from the same directory, without having to issue a second `cd` command. Why doesn't HTTP support this?

State would make individual HTTP requests simpler, but it would make the HTTP protocol much more complicated. An FTP client is much more complicated than an HTTP client, precisely because the session state must be kept in sync between client and server. This is a complex task even over a reliable network, which the Internet is not.

To eliminate state from a protocol is to eliminate a lot of failure conditions. The server never has to worry about the client timing out, because no interaction lasts longer than

Spotify USA Inc. Exhibit 1014 Page 80

a single request. The server never loses track of "where" each client is in the application, because the client sends all neccessary information with each request. The client never ends up performing an action in the wrong "working directory" due to the server keeping some state around without telling the client.

Statelessness also brings new features. It's easier to distribute a stateless application across load-balanced servers. Since no two requests depend on each other, they can be handled by two different servers that never coordinate with each other. Scaling up is as simple as plugging more servers into the load balancer. A stateless application is also easy to cache: a piece of software can decide whether or not to cache the result of an HTTP request just by looking at that one request. There's no nagging uncertainty that state from a previous request might affect the cacheability of this one.

The client benefits from statelessness as well. A client can process the "mice" directory up to page 50, bookmark `/search?q=mice&start=50`, and come back a week later without having to grind through dozens of predecessor states. A URI that works when you're hours deep into an HTTP session will work the same way as the first URI sent in a new session.

To make your service addressabile you have to put in some work, dissect your application's data into sets of resources. HTTP is an intrinsically stateless protocol, so when you write web services, you get statelessness by default. You have to do something to break it.

The most common way to break statelessness is to use your framework's version of HTTP sessions. The first time a user visits your site, he gets a unique string that identifies his session with the site. The string may be kept in a cookie, or the site may propagate a unique string through all the URIs it serves a particular client. Here's an session cookie being set by a Rails application:

```
Set-Cookie: _session_id=c1c934bbe6168dcb904d21a7f5644a2d; path=/
```

This URI propagates the session ID in a PHP application: `http://www.example.com/forums?PHPSESSID=27314962133`.

The important thing is, that nonsensical hex or decimal number is not the state. It's a key into a data structure on the server, and the data structure contains the state. There's nothing unRESTful about stateful URIs: that's how the server communicates possible next states to the client. However, there *is* something unRESTful about cookies, as I discuss in "The Trouble with Cookies." To use a web browser analogy, cookies break a web service client's back button.

Think of the query variable `start=10` in a URI, embedded in an HTML page served by the Google search engine. That's the server sending a possible next state to the client.

But those URIs need to *contain* the state, not just provide a key to state stored on the server. `start=10` means something on its own, and `PHPSESSID=27314962133` doesn't. RESTfulness requires that the state stay on the client side, and be transmitted to the

Spotify USA Inc. Exhibit 1014 Page 81

server for every request that needs it. The server can nudge the client toward new states, by sending stateful links for the client to follow, but it can't keep any state of its own.

Application State Versus Resource State

When we talk about "statelessness," what counts as "state"? What's the difference between persistent data, the useful server-side data that makes us want to use web services in the first place, and this state we're trying to keep off the server? The Flickr web service lets you upload pictures to your account, and those pictures are stored on the server. It would be crazy to make the client send every one of its pictures along with every request to `flickr.com`, just to keep the server from having to store any state. That would defeat the whole point of the service. But what's the difference between this scenario, and state about the client's session, which I claim should be kept off the server?

The problem is one of terminology. Statelessness implies there's only one kind of state and that the server should go without it. Actually, there are two kinds of state. From this point on in the book I'm going to distinguish between *application state*, which ought to live on the client, and *resource state*, which ought to live on the server.

When you use a search engine, your current query and your current page are bits of client state. This state is different for every client. You might be on page 3 of the search results for "jellyfish," and I might be on page 1 of the search results for "mice." The page number and the query are different because we took different paths through the application. Our respective clients store different bits of application state.

A web service only needs to care about your application state when you're actually making a request. The rest of the time, it doesn't even know you exist. This means that whenever a client makes a request, it must include all the application states the server will need to process it. The server might send back a page with links, telling the client about other requests it might want to make in the future, but then it can forget all about the client until the next request. That's what I mean when I say a web service should be "stateless." The client should be in charge of managing its own path through the application.

Resource state is the same for every client, and its proper place is on the server. When you upload a picture to Flickr, you create a new resource: the new picture has its own URI and can be the target of future requests. You can fetch, modify, and delete the "picture" resource through HTTP. It's there for everybody: I can fetch it too. The picture is a bit of resource state, and it stays on the server until a client deletes it.

Client state can show up when you don't expect it. Lots of web services make you sign up for a unique string they call an API key or application key. You send in this key with every request, and the server restricts uses it to restrict you to a certain number of requests a day. For instance, an API key for Google's deprecated SOAP search API is good for 1,000 requests a day. That's client state: it's different for every client. Once you exceed the limit, the behavior of the service changes dramatically: on request 1,000

Spotify USA Inc. Exhibit 1014 Page 82

you get your data, and on request 1,001 you get an error. Meanwhile, I'm on request 402 and the service still works fine for me.

Of course, clients can't be trusted to self-report this bit of application state: the temptation to cheat is too great. So servers keep this kind of application state on the server, violating statelessness. The API key is like the Rails `_session_id` cookie, a key into a server-side client session that lasts one day. This is fine as far as it goes, but there's a scalability price to be paid. If the service is to be distributed across multiple machines, every machine in the cluster needs to know that you're on request 1,001 and I'm on request 402 (technical term: *session replication*), so that every machine knows to deny you access and let me through. Alternatively, the load balancer needs to make sure that every one of your requests goes to the same machine in the cluster (technical term: *session affinity*). Statelessness removes this requirement. As a service designer, you only need to start thinking about data replication when your *resource* state needs to be split across multiple machines.

Representations

When you split your application into resources, you increase its surface area. Your users can construct an appropriate URI and enter your application right where they need to be. But the resources aren't the data; they're just the service designer's idea of how to split up the data into "a list of open bugs" or "information about jellyfish." A web server can't send an idea; it has to send a series of bytes, in a specific file format, in a specific language. This is a *representation* of the resource.

A resource is a source of representations, and a representation is just some data about the current state of a resource. Most resources are themselves items of data (like a list of bugs), so an obvious representation of a resource is the data itself. The server might present a list of open bugs as an XML document, a web page, or as comma-separated text. The sales numbers for the last quarter of 2004 might be represented numerically or as a graphical chart. Lots of news sites make their articles available in an ad-laden format, and in a stripped-down "printer-friendly" format. These are all different representations of the same resources.

But some resources represent physical objects, or other things that can't be reduced to information. What's a good representation for such things? You don't need to worry about perfect fidelity: a representation is *any useful information* about the state of a resource.

Consider a physical object, a soda machine, hooked up to a web service.[‖] The goal is to let the machine's customers avoid unneccessary trips to the machine. With the service, customers know when the soda is cold, and when their favorite brand is sold out.

[‖] This idea is based on the CMU Coke machine (*http://www.cs.cmu.edu/%7Ecoke/*), which for many years was observed by instruments and whose current state was accessible through the Finger protocol. The machine is still around, though at the time of writing its state was not accessible online.

Spotify USA Inc. Exhibit 1014 Page 83

Nobody expects the physical cans of soda to be made available through the web service, because physical objects aren't data. But they do have data *about* them: metadata. Each slot in the soda machine can be instrumented with a device that knows about the flavor, price, and temperature of the next available can of soda. Each slot can be exposed as a resource, and so can the soda machine as a whole. The metadata from the instruments can be used in representations of the resources.

Even when one of an object's representations contains the actual data, it may also have representations that contain metadata. An online bookstore may serve two representations of a book:

1. One containing only metadata, like a cover image and reviews, used to advertise the book.
2. An electronic copy of the data in the book, sent to you via HTTP when you pay for it.

Representations can flow the other way, too. You can send a representation of a new resource to the server and have the server create the resource. This is what happens when you upload a picture to Flickr. Or you can give the server a new representation of an existing resource, and have the server modify the resource to bring it in line with the new representation.

Deciding Between Representations

If a server provides multiple representations of a resource, how does it figure out which one the client is asking for? For instance, a press release might be put out in both English and Spanish. Which one does a given client want?

There are a number of ways to figure this out within the constraints of REST. The simplest, and the one I recommend for the Resource-Oriented Architecture, is to give a distinct URI to each representation of a resource. *http://www.example.com/releases/104.en* could designate the English representation of the press release, and *http://www.example.com/releases/104.es* could designate the Spanish representation.

I recommend this technique for ROA applications because it means the URI contains all information neccessary for the server to fulfill the request. The disadvantage, as whenever you expose multiple URIs for the same resource, is dilution: people who talk about the press release in different languages appear to be talking about different things. You can mitigate this problem somewhat by exposing the URI *http://www.example.com/releases/104* to mean the release as a Platonic form, independent of any language.

The alternative way is called *content negotiation*. In this scenario the only exposed URI is the Platonic form URI, *http://www.example.com/releases/104*. When a client makes a request for that URI, it provides special HTTP request headers that signal what kind of representations the client is willing to accept.

Spotify USA Inc. Exhibit 1014 Page 84

Your web browser has a setting for language preferences: which languages you'd prefer to get web pages in. The browser submits this information with every HTTP request, in the `Accept-Language` header. The server usually ignores this information because most web pages are available in only one language. But it fits with what we're trying to do here: expose different-language representations of the same resource. When a client requests *http://www.example.com/releases/104*, the server can decide whether to serve the English or the Spanish representation based on the client's `Accept-Language` header.



The Google search engine is a good place to try this out. You can get your search results in almost any language by changing your browser language settings, or by manipulating the `hl` query variable in the URI (for instance, `hl=tr` for Turkish). The search engine supports both content negotiation and different URIs for different representations.

A client can also set the `Accept` header to specify which file format it prefers for representations. A client can say it prefers XHTML to HTML, or SVG to any other graphics format.

The server is allowed to use any of this request metadata when deciding which representation to send. Other types of request metadata include payment information, authentication credentials, the time of the request, caching directives, and even the IP address of the client. All of these might make a difference in the server's decision of what data to include in the representation, which language and which format to use, and even whether to send a representation at all or to deny access.

It's RESTful to keep this information in the HTTP headers, and it's RESTful to put it in the URI. I recommend keeping as much of this information as possible in the URI, and as little as possible in request metadata. I think URIs are more useful than metadata. URIs get passed around from person to person, and from program to program. The request metadata almost always gets lost in transition.

Here's a simple example of this dilemma: the W3C HTML validator, a web service available at *http://validator.w3.org/*. Here's a URI to a resource on the W3C's site, a validation report on the English version of my hypothetical press release: `http://validator.w3.org/check?uri=http%3A%2F%2Fwww.example.com%2Freleases%2F104.en`.

Here's another resource: a validation report on the Spanish version of the press release: `http://validator.w3.org/check?uri=http%3A%2F%2Fwww.example.com%2Freleases%2F104.es`.

Every URI in your web space becomes a resource in the W3C's web application, whether or not it designates a distinct resource on your site. If your press release has a separate URI for each representation, you can get two resources from the W3C: validation reports for the English and the Spanish versions of the press release.

But if you only expose the Platonic form URI, and serve both representations from that URI, you can only get one resource from the W3C. That would be a validation report

Spotify USA Inc. Exhibit 1014 Page 85

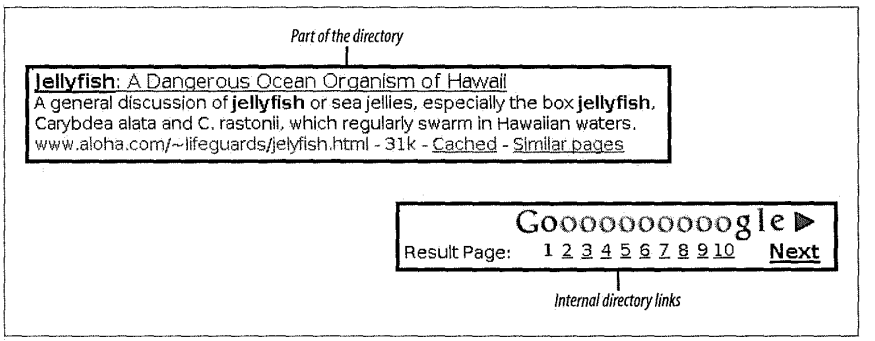


Figure 4-3. Closeup on a page of Google search results

for the *default* version of the press release (probably the English one). You've got no way of knowing whether or not the Spanish representation contains HTML formatting errors. If the server doesn't expose the Spanish press release as its own URI, there's no corresponding resource available on the W3C site. This doesn't mean you can't expose that Platonic form URI: just that it shouldn't be the only URI you use.

Unlike humans, computer programs are very bad at dealing with representations they didn't expect. I think an automated web client should be as explicit as possible about the representation it wants. This almost always means specifying a representation in the URL.

Links and Connectedness

Sometimes representations are little more than serialized data structures. They're intended to be sucked of their data and discarded. But in the most RESTful services, representations are hypermedia: documents that contain not just data, but links to other resources.

Let's take the search example again. If you go to Google's directory of documents about jellyfish (*http://www.google.com/search?q=jellyfish*), you see some search results, and a set of internal links to other pages of the directory. Figure 4-3 shows a representative sample of the page.

There's data here, and links. The data says that somewhere on the Web, someone said such-and-such about jellyfish, with emphasis on two species of Hawaiian jellyfish. The links give you access to other resources: some within the Google search "web service," and some elsewhere on the Web:

- The external web page that talks about jellyfish: `http://www.aloha.com/~lifeguards/jelyfish.html`. The main point of this web service, of course, is to present links of this sort.

Spotify USA Inc. Exhibit 1014 Page 86

- A link to a Google-provided cache of the extrenal page (the "Cached" link). These links always have long URIs that point to Google-owned IP addresses, like `http://209.85.165.104/search?q=cache:FQrLzPUOtKQJ...`
- A link to a directory of pages Google thinks are related to the external page (*http://www.google.com/search?q=related:www.aloha.com/~lifeguards/jelyfish.html*, linked as "Similar pages"). This is another case of a web service taking a URI as input.
- A set of navigation links that take you to different pages of the "jellyfish" directory: *http://www.google.com/search?q=jellyfish&start=10*, *http://www.google.com/search?q=jellyfish&start=20*, and so on.

Earlier in this chapter, I showed what might happen if HTTP was a stateful protocol like FTP. Figure 4-2 shows the paths a stateful HTTP client might take during a "session" with `www.google.com`. HTTP doesn't really work that way, but that figure does a good job of showing how we use the human web. To use a search engine we start at the home page, fill out a form to do a search, and then click links to go to subsequent pages of results. We don't usually type in one URI after another: we follow links and fill out forms.

If you've read about REST before, you might have encountered an axiom from the Fielding dissertation: "Hypermedia as the engine of application state." This is what that axiom means: the current state of an HTTP "session" is not stored on the server as a resource state, but tracked by the client as an application state, and created by the path the client takes through the Web. The server guides the client's path by serving "hypermedia": links and forms inside hypertext representations.

The server sends the client guidelines about which states are near the current one. The "next" link on *http://www.google.com/search?q=jellyfish* is a *lever of state*: it shows you how to get from the current state to a related one. This is very powerful. A document that contains a URI points to another possible state of the application: "page two," or "related to this URI," or "a cached version of this URI." Or it may be pointing to a possible state of a totally different application.

I'm calling the quality of having links *connectedness*. A web service is connected to the extent that you can put the service in different states just by following links and filling out forms. I'm calling this "connectedness" because "hypermedia as the engine of application state" makes the concept sound more difficult than it is. All I'm saying is that resources should link to each other in their representations.

The human web is easy to use because it's well connected. Any experienced user knows how to type URIs into the browser's address bar, and how to jump around a site by modifying the URI, but many users do all their web surfing from a single starting point: the browser home page set by their ISP. This is possible because the Web is well connected. Pages link to each other, even across sites.

Spotify USA Inc. Exhibit 1014 Page 87

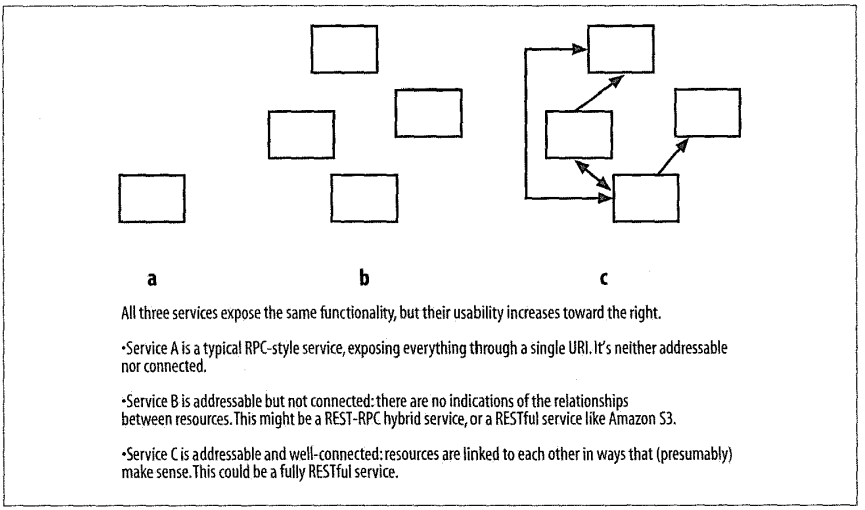


Figure 4-4. One service three ways

But most web services are not internally connected, let alone connected to each other. Amazon S3 is a RESTful web service that's addressible and stateless, but not connected. S3 representations never include URIs. To GET an S3 bucket, you have to know the rules for constructing the bucket's URI. You can't just GET the bucket list and follow a link to the bucket you want.

Example 4-1 shows an S3 bucket list that I've changed (I added a `URI` tag) so that it's connected. Compare to Example 3-5, which has no `URI` tag. This is just one way of introducing URIs into an XML representation. As resources become better-connected, the relationships between them becomes more obvious (see Figure 4-4).

Example 4-1. A connected "list of your buckets"

```
<?xml version='1.0' encoding='UTF-8'?>
<ListAllMyBucketsResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
 <Owner>
  <ID>c0363f7260f2f5fcf38d48039f4fb5cab21b060577817310be5170e7774aad70</ID>
  <DisplayName>leonardr28</DisplayName>
 </Owner>
 <Buckets>
  <Bucket>
   <Name>crummy.com</Name>
   <URI>https://s3.amazonaws.com/crummy.com</URI>
   <CreationDate>2006-10-26T18:46:45.000Z</CreationDate>
  </Bucket>
 </Buckets>
</ListAllMyBucketsResult>
```


Spotify USA Inc. Exhibit 1014 Page 88

The Uniform Interface

All across the Web, there are only a few basic things you can do to a resource. HTTP provides four basic methods for the four most common operations:

- Retrieve a representation of a resource: *HTTP GET*
- Create a new resource: *HTTP PUT* to a new URI, or *HTTP POST* to an existing URI (see the "POST" section below)
- Modify an existing resource: *HTTP PUT* to an existing URI
- Delete an existing resource: *HTTP DELETE*

I'll explain how these four are used to represent just about any operation you can think of. I'll also cover two HTTP methods for two less common operations: HEAD and OPTIONS.

GET, PUT, and DELETE

These three should be familiar to you from the S3 example in Chapter 3. To fetch or delete a resource, the client just sends a GET or DELETE request to its URI. In the case of a GET request, the server sends back a representation in the response entity-body. For a DELETE request, the response entity-body may contain a status message, or nothing at all.

To create or modify a resource, the client sends a PUT request that usually includes an entity-body. The entity-body contains the client's proposed new representation of the resource. What data this is, and what format it's in, depends on the service. Whatever it looks like, this is the point at which application state moves onto the server and becomes resource state.

Again, think of the S3 service, where there are two kinds of resources you can create: buckets and objects. To create an object, you send a PUT request to its URI and include the object's content in the entity-body of your request. You do the same thing to modify an object: the new content overwrites any old content.

Creating a bucket is a little different because you don't have to specify an entity-body in the PUT request. A bucket has no resource state except for its name, and the name is part of the URI. (This is not quite true. The objects in a bucket are also elements of that bucket's resource state: after all, they're listed when you GET a bucket's representation. But every S3 object is a resource of its own, so there's no need to *manipulate* an object through its bucket. Every object exposes the uniform interface and you can manipulate it separately.) Specify the bucket's URI and you've specified its representation. PUT requests for most resources do include an entity-body containing a representation, but as you can see it's not a requirement.

Spotify USA Inc. Exhibit 1014 Page 89

HEAD and OPTIONS

There are three other HTTP methods I consider part of the uniform interface. Two of them are simple utility methods, so I'll cover them first.

- Retrieve a metadata-only representation: *HTTP HEAD*
- Check which HTTP methods a particular resource supports: *HTTP OPTIONS*

saw the HEAD method exposed by the S3 services's resources in Chapter 3. An S3 client uses HEAD to fetch metadata about a resource without downloading the possibly enormous entity-body. That's what HEAD is for. A client can use HEAD to check whether a resource exists, or find out other information about the resource, without fetching its entire representation. HEAD gives you exactly what a GET request would give you, but without the entity-body.



There are two standard HTTP methods I don't cover in this book: TRACE and CONNECT. TRACE is used to debug proxies, and CONNECT is used to forward some other protocol through an HTTP proxy.

The OPTIONS method lets the client discover what it's allowed to do to a resource. The response to an OPTIONS request contains the HTTP `Allow` header, which lays out the subset of the uniform interface this resource supports. Here's a sample `Allow` header:

```
Allow: GET, HEAD
```

That particular header means the client can expect the server to act reasonably to a GET or HEAD request for this resource, but that the resource doesn't support any other HTTP methods. Effectively, this resource is read-only.

The headers the client sends in the request may affect the `Allow` header the server sends in response. For instance, if you send a proper `Authorization` header along with an OPTIONS request, you may find that you're allowed to make GET, HEAD, PUT, and DELETE requests against a particular URI. If you send the same OPTIONS request but omit the `Authorization` header, you may find that you're only allowed to make GET and HEAD requests. The OPTIONS method lets the client do simple access control checks.

In theory, the server can send additional information in response to an OPTIONS request, and the client can send OPTIONS requests that ask very specific questions about the server's capabilities. Very nice, except there are no accepted standards for what a client might ask in an OPTIONS request. Apart from the `Allow` header there are no accepted standards for what a server might send in response. Most web servers and frameworks feature very poor support for OPTIONS. So far, OPTIONS is a promising idea that nobody uses.

Spotify USA Inc. Exhibit 1014 Page 90

POST

Now we come to that most misunderstood of HTTP methods: POST. This method essentially has two purposes: one that fits in with the constraints of REST, and one that goes outside REST and introduces an element of the RPC style. In complex cases like this it's best to go back to the original text. Here's what RFC 2616, the HTTP standard, says about POST (this is from section 9.5):

> POST is designed to allow a uniform method to cover the following functions:
>
> - Annotation of existing resources;
> - Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;
> - Providing a block of data, such as the result of submitting a form, to a data-handling process;
> - Extending a database through an append operation.
>
> The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

What does this mean in the context of REST and the ROA?

Creating subordinate resources

In a RESTful design, POST is commonly used to create subordinate resources: resources that exist in relation to some other "parent" resource. A weblog program may expose each weblog as a resource (`/weblogs/myweblog`), and the individual weblog entries as subordinate resources (`/weblogs/myweblog/entries/1`). A web-enabled database may expose a table as a resource, and the individual database rows as its subordinate resources. To create a weblog entry or a database record, you POST to the parent: the weblog or the database table. What data you post, and what format it's in, depends on the service, but as with PUT, this is the point where application state becomes resource state. You may see this use of POST called *POST(a)*, for "append". When I say "POST" in this book, I almost always mean POST(a).

Why can't you just use PUT to create subordinate resources? Well, sometimes you can. An S3 object is a subordinate resource: every S3 object is contained in some S3 bucket. But we don't create an S3 object by sending a POST request to the bucket. We send a PUT request directly to the URI of the object. The difference between PUT and POST is this: the client uses PUT when it's in charge of deciding which URI the new resource should have. The client uses POST when the *server* is in charge of deciding which URI the new resource should have.

The S3 service expects clients to create S3 objects with PUT, because an S3 object's URI is completely determined by its name and the name of the bucket. If the client

Spotify USA Inc. Exhibit 1014 Page 91

knows enough to create the object, it knows what its URI will be. The obvious URI to use as the target of the PUT request is the one the bucket will live at once it exists.

But consider an application in which the server has more control over the URIs: say, a weblog program. The client can gather all the information neccessary to create a weblog entry, and still not know what URI the entry will have once created. Maybe the server bases the URIs on ordering or an internal database ID: will the final URI be `/weblogs/myweblog/entries/1` or `/weblogs/myweblog/entries/1000`? Maybe the final URI is based on the posting time: what time does the server think it is? The client shouldn't have to know these things.

The POST method is a way of creating a new resource without the client having to know its exact URI. In most cases the client only needs to know the URI of a "parent" or "factory" resource. The server takes the representation from the entity-body and use it to create a new resource "underneath" the "parent" resource (the meaning of "underneath" depends on context).

The response to this sort of POST request usually has an HTTP status code of 201 ("Created"). Its `Location` header contains the URI of the newly created subordinate resource. Now that the resource actually exists and the client knows its URI, future requests can use the PUT method to modify that resource, GET to fetch a representation of it, and DELETE to delete it.

Table 4-1 shows how a PUT request to a URI might create or modify the underlying resource; and how a POST request to the same URI might create a new, subordinate resource.

Table 4-1. PUT actions

	PUT to a new resource	PUT to an existing resource	POST
`/weblogs`	N/A (resource already exists)	No effect	Create a new weblog
`/weblogs/myweblog`	Create this weblog	Modify this weblog's settings	Create a new weblog entry
`/weblogs/myweblog/entries/1`	N/A (how would you get this URI?)	Edit this weblog entry	Post a comment to this weblog entry

Appending to the resource state

The information conveyed in a POST to a resource doesn't have to result in a whole new subordinate resource. Sometimes when you POST data to a resource, it appends the information you POSTed to its own state, instead of putting it in a new resource.

Consider an event logging service that exposes a single resource: the log. Say its URI is `/log`. To get the log you send a GET request to `/log`.

Now, how should a client append to the log? The client might send a PUT request to `/log`, but the PUT method has the implication of creating a new resource, or

Spotify USA Inc. Exhibit 1014 Page 92

overwriting old settings with new ones. The client isn't doing either: it's just appending information to the end of the log.

The POST method works here, just as it would if each log entry was exposed as a separate resource. The semantics of POST are the same in both cases: the client adds subordinate information to an existing resource. The only difference is that in the case of the weblog and weblog entries, the subordinate information showed up as a new resource. Here, the subordinate information shows up as new data in the parent's representation.

Overloaded POST: The not-so-uniform interface

That way of looking at things explains most of what the HTTP standard says about POST. You can use it to create resources underneath a parent resource, and you can use it to append extra data onto the current state of a resource. The one use of POST I haven't explained is the one you're probably most familiar with, because it's the one that drives almost all web applications: providing a block of data, such as the result of submitting a form, to a data-handling process.

What's a "data-handling process"? That sounds pretty vague. And, indeed, just about anything can be a data-handling process. Using POST this way turns a resource into a tiny message processor that acts like an XML-RPC server. The resource accepts POST requests, examines the request, and decides to do... something. Then it decides to serve to the client... some data.

I call this use of POST *overloaded POST*, by analogy to operator overloading in a programming language. It's overloaded because a single HTTP method is being used to signify any number of non-HTTP methods. It's confusing for the same reason operator overloading can be confusing: you thought you knew what HTTP POST did, but now it's being used to achieve some unknown purpose. You might see overloaded POST called *POST(p)*, for "process."

When your service exposes overloaded POST, you reopen the question: "why should the server do this instead of that?" Every HTTP request has to contain method information, and when you use overloaded POST it can't go into the HTTP method. The POST method is just a directive to the server, saying: "Look inside the HTTP request for the real method information." The real information may be in the URI, the HTTP headers, or the entity-body. However it happens, an element of the RPC style has crept into the service.

When the method information isn't found in the HTTP method, the interface stops being uniform. The real method information might be anything. As a REST partisan I don't like this very much, but occasionally it's unavoidable. By Chapter 9 you'll have seen how just about any scenario you can think of can be exposed through HTTP's uniform interface, but sometimes the RPC style is the easiest way to express complex operations that span multiple resources.

Spotify USA Inc. Exhibit 1014 Page 93

You may need to expose overloaded POST even if you're only using POST to create subordinate resources or to append to a resource's representation. What if a single resource supports both kinds of POST? How does the server know whether a client is POSTing to create a subordinate resource, or to append to the existing resource's representation? You may need to put some additional method information elsewhere in the HTTP request.

Overloaded POST should not be used to cover up poor resource design. Remember, a resource can be anything. It's usually possible to shuffle your resource design so that the uniform interface applies, rather than introduce the RPC style into your service.

Safety and Idempotence

If you expose HTTP's uniform interface as it was designed, you get two useful properties for free. When correctly used, GET and HEAD requests are *safe*. GET, HEAD, PUT and DELETE requests are *idempotent*.

Safety

A GET or HEAD request is a request to read some data, not a request to change any server state. The client can make a GET or HEAD request 10 times and it's the same as making it once, or never making it at all. When you GET *http://www.google.com/search?q=jellyfish*, you aren't changing anything about the directory of jellyfish resources. You're just retrieving a representation of it. A client should be able to send a GET or HEAD request to an unknown URI and feel safe that nothing disastrous will happen.

This is not to say that GET and HEAD requests can't have side effects. Some resources are hit counters that increment every time a client GETs them. Most web servers log every incoming request to a log file. These are side effects: the server state, and even the resource state, is changing in response to a GET request. But the client didn't ask for the side effects, and it's not responsible for them. A client should never make a GET or HEAD request just for the side effects, and the side effects should never be so big that the client might wish it hadn't made the request.

Idempotence

Idempotence is a slightly tricker notion. The idea comes from math, and if you're not familiar with idempotence, a math example might help. An idempotent operation in math is one that has the same effect whether you apply it once, or more than once. Multiplying a number by zero is idempotent: $4 \times 0 \times 0 \times 0$ is the same as 4×0.[#] By analogy, an operation on a resource is idempotent if making one request is the same as

[#] Multiplying a number by one is both safe and idempotent: $4 \times 1 \times 1 \times 1$ is the same as 4×1, which is the same as 4. Multiplication by zero is not safe, because 4×0 is not the same as 4. Multiplying by any other number is neither safe nor idempotent.

Spotify USA Inc. Exhibit 1014 Page 94

making a series of identical requests. The second and subsequent requests leave the resource state in exactly the same state as the first request did.

PUT and DELETE operations are idempotent. If you DELETE a resource, it's gone. If you DELETE it again, it's still gone. If you create a new resource with PUT, and then resend the PUT request, the resource is still there and it has the same properties you gave it when you created it. If you use PUT to change the state of a resource, you can resend the PUT request and the resource state won't change again.

The practical upshot of this is that you shouldn't allow your clients to PUT representations that change a resource's state in relative terms. If a resource keeps a numeric value as part of its resource state, a client might use PUT to set that value to 4, or 0, or 50, but not to increment that value by 1. If the initial value is 0, sending two PUT requests that say "set the value to 4" leaves the value at 4. If the initial value is 0, sending two PUT requests that say "increment the value by 1" leaves the value not at 1, but at 2. That's not idempotent.

Why safety and idempotence matter

Safety and idempotence let a client make reliable HTTP requests over an unreliable network. If you make a GET request and never get a response, just make another one. It's safe: even if your earlier request went through, it didn't have any real effect on the server. If you make a PUT request and never get a response, just make another one. If your earlier request got through, your second request will have no additional effect.

POST is neither safe nor idempotent. Making two identical POST requests to a "factory" resource will probably result in two subordinate resources containing the same information. With overloaded POST, all bets are off.

The most common misuse of the uniform interface is to expose unsafe operations through GET. The del.icio.us and Flickr APIs both do this. When you GET *https://api.del.icio.us/posts/delete*, you're not fetching a representation: you're modifying the del.icio.us data set.

Why is this bad? Well, here's a story. In 2005 Google released a client-side caching tool called Web Accelerator. It runs in conjunction with your web browser and "pre-fetches" the pages linked to from whatever page you're viewing. If you happen to click one of those links, the page on the other side will load faster, because your computer has already fetched it.

Web Accelerator was a disaster. Not because of any problem in the software itself, but because the Web is full of applications that misuse GET. Web Accelerator assumed that GET operations were safe, that clients could make them ahead of time just in case a human being wanted to see the corresponding representations. But when it made those GET requests to real URIs, it changed the data sets. People lost data.

There's plenty of blame to go around: programmers shouldn't expose unsafe actions through GET, and Google shouldn't have released a real-world tool that didn't work

Spotify USA Inc. Exhibit 1014 Page 95

with the real-world web. The current version of Web Accelerator ignores all URIs that contain query variables. This solves part of the problem, but it also prevents many resources that are safe to use through GET (such as Google web searches) from being pre-fetched.

Multiply the examples if you like. Many web services and web applications use URIs as input, and the first thing they do is send a GET request to fetch a representation of a resource. These services don't mean to trigger catastrophic side effects, but it's not up to them. It's up to the service to handle a GET request in a way that complies with the HTTP standard.

Why the Uniform Interface Matters

The important thing about REST is not that you use the specific uniform interface that HTTP defines. REST specifies a uniform interface, but it doesn't say *which* uniform interface. GET, PUT, and the rest are not a perfect interface for all time. What's important is the uniformity: that every service use HTTP's interface the same way.

The point is not that GET is the best name for a read operation, but that GET means "read" across the Web, no matter which resource you're using it on. Given a URI of a resource, there's no question of how you get a representation: you send an HTTP GET request to that URI. The uniform interface makes any two services as similar as any two web sites. Without the uniform interface, you'd have to learn how each service expected to receive and send information. The rules might even be different for different resources within a single service.

You can program a computer to understand what GET means, and that understanding will apply to every RESTful web service. There's not much to understand. The service-specific code can live in the handling of the representation. Without the uniform interface, you get a multiplicity of methods taking the place of GET: `doSearch` and `getPage` and `nextPrime`. Every service speaks a different language. This is also the reason I don't like overloaded POST very much: it turns the simple Esperanto of the uniform interface into a Babel of one-off sublanguages.

Some applications extend HTTP's uniform interface. The most obvious case is WebDAV, which adds eight new HTTP methods including MOVE, COPY, and SEARCH. Using these methods in a web service would not violate any precept of REST, because REST doesn't say what the uniform interface should look like. Using them would violate my Resource-Oriented Architecture (I've explicitly tied the ROA to the standard HTTP methods), but your service could still be resource-oriented in a general sense.

The real reason not to use the WebDAV methods is that doing so makes your service incompatible with other RESTful services. Your service would use a different uniform interface than most other services. There are web services like Subversion that use the WebDAV methods, so your service wouldn't be all alone. But it would be part of a much smaller web. This is why making up your own HTTP methods is a very, very bad

Spotify USA Inc. Exhibit 1014 Page 96

idea: your custom vocabulary puts you in a community of one. You might as well be using XML-RPC.

Another uniform interface consists solely of HTTP GET and overloaded POST. To fetch a representation of a resource, you send GET to its URI. To create, modify, or delete a resource, you send POST. This interface is perfectly RESTful, but, again, it doesn't conform to my Resource-Oriented Architecture. This interface is just rich enough to distinguish between safe and unsafe operations. A resource-oriented web application would use this interface, because today's HTML forms only support GET and POST.

That's It!

That's the Resource-Oriented Architecture. It's just four concepts:

1. Resources
2. Their names (URIs)
3. Their representations
4. The links between them

and four properties:

1. Addressability
2. Statelessness
3. Connectedness
4. A uniform interface

Of course, there are still a lot of open questions. How should a real data set be split into resources, and how should the resources be laid out? What should go into the actual HTTP requests and responses? I'm going to spend much of the rest of the book exploring issues like these.

Spotify USA Inc. Exhibit 1014 Page 97

Web Services/Web Development

O'REILLY®

RESTful Web Services



"Every developer working with the Web needs to read this book."

—David Heinemeier Hansson, creator of the Rails framework

"RESTful Web Services *finally provides a practical road map for constructing services that embrace the Web, instead of trying to route around it."*

—Adam Trachtenberg, PHP author and eBay Web Services Evangelist

You've built web sites that can be used by other people. But can you build web sites that are usable by machines? The World Wide Web isn't just a big distributed application—it's a *platform* for distributed computing. Web services, mashups, and Ajax applications all use the Web as a platform. But today's web service technologies have lost sight of the simplicity that made the Web successful; they don't work like the Web, and they're missing out on its advantages.

This book puts the "Web" back into web services. It explains how you can connect to the programmable web with the technologies you already use every day. The key is REST, the architectural style that drives the Web. *RESTful Web Services*:

- Emphasizes the power of basic web technologies—the HTTP application protocol, the URI naming standard, and the XML markup language
- Introduces the Resource-Oriented Architecture (ROA), a common-sense set of rules for designing RESTful web services
- Reveals how a RESTful design is simpler, more versatile, and more scalable than a design based on Remote Procedure Calls (RPCs)
- Includes real-world examples of RESTful web services, like Amazon's Simple Storage Service and the Atom Publishing Protocol
- Discusses web service clients for popular programming languages
- Demonstrates how to implement RESTful services in three popular frameworks—Ruby on Rails, Restlet (for Java), and Django (for Python)
- Focuses on practical issues, such as how to design and implement RESTful web services and clients

RESTful Web Services is the first book that applies the REST design philosophy to real web services. It sets down the best practices you need to make your design a success and the techniques you need to turn your design into working code. You can harness the power of the Web for programmable applications—you just have to work with the Web instead of against it. This book explains how.

X004LMBIYL
New - RESTful Web Services

Safari® BOOKS ONLINE ENABLED
Includes FREE 45-Day Online Edition