RESTful Web Services（影印版）

RESTful Web Services

O'REILLY®
東南大學出版社

Leonard Richardson & Sam Ruby 著
David Heinemeier Hansson 序

RESTful Web Services（影印版）

RESTful Web Services

Leonard Richardson & Sam Ruby

O'REILLY®

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

O'Reilly Media, Inc. 授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

REST架构的网络服务＝RESTful Web Services：英文/（美）理查森（Richardson, L.），（美）鲁比（Ruby, S.）著. —影印本. —南京：东南大学出版社，2007.11

书名原文：RESTful Web Services

ISBN 978-7-5641-0960-8

Ⅰ.R… Ⅱ.①理… ②鲁… Ⅲ.互联网络－网络服务器－程序设计－英文 Ⅳ.TP368.5

中国版本图书馆CIP数据核字（2007）第154488号

江苏省版权局著作权合同登记

图字：10-2007-194号

书　　名/ RESTful Web Services
责任编辑/ 张烨
封面设计/ Karen Montgomery，张健
出版发行/ 东南大学出版社（press.seu.edu.cn）
地　　址/ 南京四牌楼2号（邮政编码210096）
印　　刷/ 扬中市印刷有限公司
开　　本/ 787毫米×980毫米　16开本　28印张　470千字
版　　次/ 2007年11月第1版　2007年11月第1次印刷
印　　数/ 0001-4000册
书　　号/ ISBN 978-7-5641-0960-8/TP·156
定　　价/ 46.00元（册）

RESTful Web Services（影印版）

RESTful Web Servi ces

About the Authors

Leonard Richardson (*http://www.crummy.com/*) is the author of the *Ruby Cookbook* (O'Reilly) and of several open source libraries, including Beautiful Soup. A California native, he currently lives in New York.

Sam Ruby is a prominent software developer who has made significant contributions to many Apache Software Foundation open source projects, and to the standardization of web feeds via his involvement with the Atom web feed standard and the popular Feed Validator web service. He currently holds a Senior Technical Staff Member position in the Emerging Technologies Group of IBM. He resides in Raleigh, North Carolina.

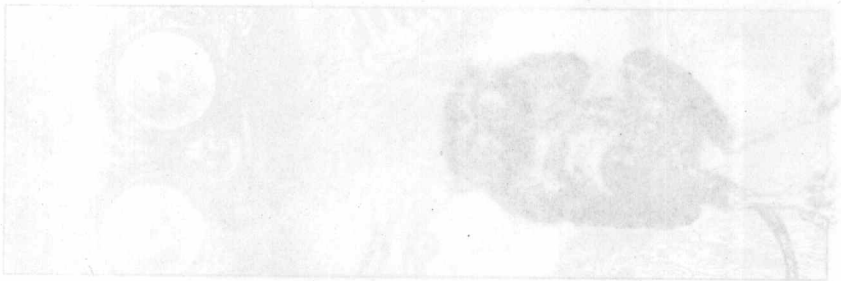
Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *RESTful Web Services* is a vulpine phalanger (*P. vulpina*). Phalanger is the general term given to animals of the Phalangeridae family, which includes possums and cuscuses. (One should not confuse the Australian possum with the American opossum; they are both marsupials, but very different.) The term phalanger is derived from the Greek word phalanges, which means finger or toe bone. The omnivorous phalanger uses its claw-fingered paws (with opposable thumbs) to climb, hunt, and live in trees. Phalangers are found in the forests of Australia, New Zealand, Tasmania, and some Indonesian islands. Like the most famous marsupial, the kangaroo, female phalangers carry their young around in a front pouch after birth.

Phalanger is also the name of a PHP complier project for the .NET framework.

The cover image is from *Wood's Natural History*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed.



欢迎您购买O'REILLY®的图书

您从哪一本书中得到此卡?

_____

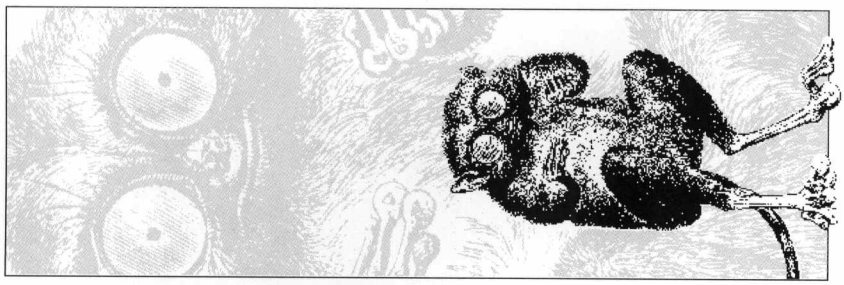
您希望引进哪方面的图书?

_____

_____

东南大学出版社O'Reilly影印图书读者俱乐部会员招募中,寄回此明信片,您将得到专属于您的会员号和相应积分。具体积分送礼方法请参见O'Reilly中国网站。

姓名	性别	年龄	影印书俱乐部会员编号: E
通讯地址			邮编
E-mail (必填)		电话	手机



O'REILLY®
http://www.oreilly.com
东南大学出版社
email: seupress@seu.edu.cn

O'Reilly Media, Inc.介绍

O'Reilly Media, Inc.是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc.知道市场上真正需要什么图书。

出版说明

随着计算机技术的成熟和广泛应用，人类正在步入一个技术迅猛发展的新时期。计算机技术的发展给人们的工业生产、商业活动和日常生活都带来了巨大的影响。然而，计算机领域的技术更新速度之快也是众所周知的，为了帮助国内技术人员在第一时间了解国外最新的技术，东南大学出版社和美国 O'Reilly Meida, Inc.达成协议，将陆续引进该公司的代表前沿技术或者在某专项领域享有盛名的著作，以影印版或者简体中文版的形式呈献给读者。其中，影印版书籍力求与国外图书“同步”出版，并且“原汁原味”展现给读者。

我们真诚地希望，所引进的书籍能对国内相关行业的技术人员、科研机构的研究人员和高校师生的学习和工作有所帮助，对国内计算机技术的发展有所促进。也衷心期望读者提出宝贵的意见和建议。

最新出版的影印版图书，包括：

- 《深入浅出 PMP》（影印版）
- 《RESTful Web Services》（影印版）
- 《学习 WCF》（影印版）
- 《学习 Ruby》（影印版）
- 《Beautiful Code》（影印版）
- 《ActionScript 3.0 精髓》（影印版）
- 《Linux 系统管理》（影印版）
- 《精通 Perl》（影印版）
- 《深入浅出面向对象分析与设计》（影印版）
- 《Linux Kernel 技术手册》（影印版）
- 《ActionScript 3.0 Cookbook》（影印版）
- 《CSS:The Missing Manual》（影印版）

For Woot, Moby, and Beet.

—Leonard

For Christopher, Catherine, and Carolyn.

—Sam

Foreword

The world of web services has been on a fast track to supernova ever since the architect astronauts spotted another meme to rocket out of pragmatism and into the universe of enterprises. But, thankfully, all is not lost. A renaissance of HTTP appreciation is building and, under the banner of REST, shows a credible alternative to what the merchants of complexity are trying to ram down everyone's throats; a simple set of principles that every day developers can use to connect applications in a style native to the Web.

RESTful Web Services shows you how to use those principles without the drama, the big words, and the miles of indirection that have scared a generation of web developers into thinking that web services are so hard that you have to rely on BigCo implementations to get anything done. Every developer working with the Web needs to read this book.

—David Heinemeier Hansson

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."

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*)

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)

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

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*)

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 how 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.

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. We show 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!