Services for a Changing World

RESTful Web APIs

Leonard Richardson &
Mike Amundsen
Foreword by Sam Ruby

O'REILLY®

RESTful Web APIs

The popularity of REST in recent years has led to tremendous growth in almost-RESTful APIs that miss out on many of the architecture's benefits. With this practical guide, you'll learn what it takes to design usable REST APIs that evolve over time. By focusing on solutions that cross a variety of domains, this book shows you how to create powerful and secure applications, using the tools designed for the world's most successful distributed computing system: the World Wide Web.

You'll explore the concepts behind REST, learn different strategies for creating hypermedia-based APIs, and then put everything together with a step-by-step guide to designing a RESTful web API.

- Examine API design strategies, including the collection pattern and pure hypermedia
- Understand how hypermedia ties representations together into a coherent API
- Discover how XMDP and ALPS profile formats can help you meet the web API "semantic challenge"
- Learn close to two-dozen standardized hypermedia data formats
- Apply best practices for using HTTP in API implementations
- Create web APIs with the JSON-LD standard and other Linked Data approaches
- Understand the CoAP protocol for using REST in embedded systems

"A terrific book! RESTful Web APIs covers the most important trends and practices in APIs today."

—John Musser
founder of ProgrammableWeb

Leonard Richardson, author of O'Reilly's *Ruby Cookbook*, has created several open source libraries, including Beautiful Soup.

Mike Amundsen has more than a dozen books to his credit, including *Building Hypermedia APIs with HTML5 and Node* (O'Reilly).

Sam Ruby is a co-chair of the W3C HTML Working Group and a Senior Technical Staff Member in the Emerging Technologies Group of IBM.

US $44.99 **CAN $51.99**
ISBN: 978-1-449-35806-8

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

Praise for *RESTful Web APIs*

"This book is the best place to start learning the essential craft of API Design."

—Matt McLarty
Cofounder, API Academy

"The entire time I read this book, I was cursing. I was cursing because as I read each explanation, I was worried that they were so good that it would be hard to find a better one to use in my own writing. You will not find another work that explores the topic so thoroughly yet explains the topic so clearly. Please, take these tools, build something fantastic, and share it with the rest of the world, okay?"

—Steve Klabnik
Author, *Designing Hypermedia APIs*

"Wonderfully thorough treatment of hypermedia formats, REST's least well understood tenet."

—Stefan Tilkov
REST evangelist, author, and consultant

"The best practical guide to hypermedia APIs. A must-have."

— Ruben Verborgh
Semantic hypermedia researcher

RESTful Web APIs

Leonard Richardson and Mike Amundsen
Foreword by Sam Ruby

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

RESTful Web APIs

by Leonard Richardson and Mike Amundsen with a Foreword by Sam Ruby

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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Meghan Blanchette
Production Editor: Christopher Hearse
Copyeditor: Jasmine Kwityn
Proofreader: Linley Dolby
Indexer: Judith McConville
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

September 2013: First Edition

Revision History for the First Edition:

2013-09-10: First release

2015-05-22: Second release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449358068* for release details.

ISBN: 978-1-449-35806-8

[LSI]

For Sienna, Dalton, and Maggie. —Leonard

For Milo "The Supervisor," my constant and patient companion throughout this and so many other projects. Thanks, buddy! —Mike

Table of Contents

Foreword

Progressive Disclosure is a concept in User Interface Design which advocates only presenting to the user the information they need when they need it. In many ways, the book you are reading right now is an example of this principle. In fact, it is quite likely that this book wouldn't have "worked" a mere seven years ago.

For you see, the programming world was quite a different place when *RESTful Web Services*, the predecessor of this book, was written. At that time, the term "REST" was was rarely used. And when it was used it was often misapplied, and widely misunderstood.

This was the case despite the fact that the standards upon which REST is based, namely HTTP and HTML, were developed and became IETF and W3C standards in roughly their current form in the second half of the 1990s. Roy Fielding's thesis paper in which he introduced the term REST and on which this book was based was itself published in 2000.

Leonard Richardson and I set out to correct this injustice. To do this, we focused primarily on the concepts underpinning HTTP, and we provided practical guidance on how to apply those concepts to applications.

I'd like to think that we helped kick a few pebbles loose that started the avalanche of support for REST that came forth since that time. REST rapidly took on a life of its own, and in the process has become a buzzword. In fact it now is pretty much the case that presenting a web interface and calling it REST is practically the default. We've definitely come a long way in a few short years.

Admittedly, REST as a term is often over applied, and not always correctly. But all things considered, I am very pleased that the concepts of resources and URIs have successfully managed to infiltrate their way into application interface design. The web, after all, is a resilient place, and these new interfaces, albeit imperfect, are leaps and bounds better than the ones that they replace.

But we can do better.

Now that those building blocks are in place, it is time to take a step back, survey the territory, and build on top of these concepts. The next logical step is to explore media types in general, and hypermedia formats in specific. While the first book focused almost exclusively on the correct application of HTTP, it is time to delve more deeply into the concepts behind hypertext media types like HTML—media types that aren't tightly bound to a single application or even a single vendor.

HTML remains a prime example of a such a hypermedia format, and it continues to hold a special place in web architecture. In fact, my personal journey of discovery has been to take a deep dive into development of the W3C standard for HTML, now branded as HTML5. And while HTML does have a prominent place in this new book, there is so much more to cover on the topic of hypermedia. So while I have remained in touch, Leonard picked up a capable replacement for my role as coauthor in Mike Amundsen.

It has been a pleasure to watch this book be written, and in reading this book I've learned about a number of media types that I had not been exposed to by any other source. More importantly, this book shows what these types have in common, and how to differentiate them, as each has its own specialty.

Hopefully the pebbles that this book kicks loose will have the same effect as its predecessor did. Who knows, perhaps in another seven years it will be time to do this all over again, and highlight some other facet of Representational State Transfer that continues to be under-appreciated.

—Sam Ruby

Introduction

> "Most software systems are created with the implicit assumption that the entire system is under the control of one entity, or at least that all entities participating within a system are acting towards a common goal and not at cross-purposes. Such an assumption cannot be safely made when the system runs openly on the Internet."
>
> — Roy Fielding
> *Architectural Styles and the Design of Network-based Software Architectures*

> "A Discordian Shall Always use the Official Discordian Document Numbering System."
>
> — Malaclypse the Younger and Lord Omar Khayyam Ravenhurst
> *Principia Discordia*

I'm going to show you a better way to do distributed computing, using the ideas underlying the most successful distributed system in history: the World Wide Web. I hope you'll read this book if you've decided (or your manager has decided) that your company needs to publish a web API. It doesn't matter whether you're planning a public API, a purely internal API, or an API accessible by trusted partners—they can all benefit from the philosophy of REST.

This is not necessarily the book for you if you want to learn how to write API *clients*. That's because most existing API designs are based on assumptions that are several years old, assumptions that I'd like to destroy.

Most of today's APIs have a big problem: once deployed, they can't change. There are big-name APIs that stay static for years at a time, as the industry changes around them, because changing them would be too difficult.

But RESTful architectures are designed for managing change. The World Wide Web is made of millions of websites, running atop thousands of different server implementations, and undergoing periodic redesigns. Websites are accessed by billions of users who are using hundreds of different client implementations on dozens of hardware platforms. Your deployment won't look like this howling mess, but the closer you come to web scale, the more familiar this picture will look.

A very simple system is always easy to change. At small scales, a RESTful system has a larger up-front design cost than a push-button solution. But as your API matures and starts to change, you'll really need some way—like REST—of adapting to change.

- An API that's commercially successful will stay available for years on end. Some APIs have hundreds or even thousands of users. Even if the problem domain only changes occasionally, the cumulative effect on clients can be huge.
- Some APIs change all the time, with new data elements and business rules constantly being added.
- In some APIs, each client can change the workflow to suit its needs. Even if the API itself never changes, each client will experience it differently.
- The people who write the API clients usually don't work on the same team as the people who write the servers. All APIs that are open to the public fall under this category. If you don't know what kind of clients are out there, you need to be very careful about making changes—or you need to have a design that can change without breaking all the clients.

If you copy existing designs for your API, you will probably only repeat the mistakes of the past. Unfortunately, most of the improvements are happening below the surface, in experiments and through slow-moving standards processes. I'll cover dozens of specific technologies in this book, including many that are still under development. But my main goal is to teach you the underlying principles of REST. Learn those, and you'll be able to exploit whichever experiments pan out and whichever standards are approved.

There are two specific problems I'm trying to solve with this book: duplication of effort and avoidance of hypermedia. Let's take a look at them.

Duplication of Effort

An API released today will be named after the company that hosts it. We talk about the "Twitter API," the "Facebook API," and the "Google+ API." These three APIs do similar things. They all have some notion of user accounts and (among other things) they all let users post a little bit of text to their accounts. But each API has a completely different design. Learning one API doesn't help you learn the next one.

Of course, Twitter, Facebook, and Google are big companies that compete with each other. They don't *want* to make it easy for you to learn their competitors' APIs. But small companies and nonprofits do the same thing. They design their APIs as though nobody else had ever had a similar idea. This interferes with their goal of getting people to actually use their APIs.

Let me show you just one example. The website ProgrammableWeb has a directory of over 8,000 APIs. As I write this, it knows about 57 microblogging APIs—APIs whose

main purpose is posting a little bit of text to a user account.[1] It's great that there are 57 companies publishing APIs in this field, but do we really need 57 different *designs*? We're not talking about something complicated here, like insurance policies or regulatory compliance. We're talking about posting a little bit of text to a user account. Do you want to be the one who designs the 58th microblogging API?

The obvious solution would be to create a standard for microblogging APIs. But there already *is* a standard that would work just fine: the Atom Publishing Protocol. It was published in 2005, and almost nobody uses it. There's something about APIs that makes everyone want to design their own from scratch, even when that makes no sense from a business perspective.

I don't think I can single-handledly stop this wasted effort, but I do think I can break down the problem into parts that make sense, and present some ways for a new API to reuse work that's already been done.

Hypermedia Is Hard

Back in 2007, Leonard Richardson and Sam Ruby wrote the predecessor to this book, *RESTful Web Services* (O'Reilly). That book also tried to address two big problems. One of the problems has been solved; the other is nowhere close to being solved.[2]

The first problem: in 2007, the REST school of API design was engaged in a standoff against a rival school that used heavyweight technologies based on SOAP and questioned the very legitimacy of the REST school. *RESTful Web Services* was a salvo in this standoff, a defense of RESTful design principles against the attacks of the SOAP school.

Well, the standoff is over, and REST won. SOAP APIs are still used, but only within the big companies that were backing the SOAP school in the first place. Pretty much all new public-facing APIs pay lip service to RESTful principles.[3]

Which brings me to the second problem: REST isn't just a technical term—it's also a marketing buzzword. For a long time, REST was a slogan that signified nothing beyond opposition to the SOAP school. Any API that didn't use SOAP was marketed as REST, even if its design made no sense or betrayed the technical principles of REST. This was inaccurate, confusing, and it gave REST—i.e., REST as a technical term—a bad name.

1. The full list of ProgrammableWeb APIs tagged with microblogging provides information about each of these APIs.
2. *RESTful Web Services* is now freely available as part of O'Reilly's Open Books Project. You can download a PDF copy of the book from the book's page.
3. If you're wondering, this is why we changed the title. The term "web services" became so tightly coupled with SOAP that when SOAP went down, it took "web services" with it. These days, everyone talks about APIs instead.

This situation has improved a lot since 2007. When I look at new APIs, I see the work of developers who understand the concepts I'll be explaining in the first few chapters of this book. Most developers who fly the REST flag today understand resources and representations, how to name resources with URLs, and how to properly use HTTP methods. The first three chapters of this book don't do much but get new developers up to speed.

But there's one aspect of REST that most developers still don't understand: hypermedia. We all understand hypermedia in the context of the Web. It's just a fancy word for links. Web pages link to each other, and the result is the World Wide Web, driven by hypermedia. But it seems we've got a mental block when it comes to hypermedia in web APIs. This is a big problem, because hypermedia is the feature that makes a web API capable of handling changes gracefully.

Starting in Chapter 4, my overriding goal for *RESTful Web APIs* will be to teach you how hypermedia works. If you've never heard of this term, I'll teach it to you along with the other important REST concepts. If you've heard of hypermedia but the concept intimidates you, I'll do what I can to build up your courage. If you just haven't been able to wrap your head around hypermedia, I'll show it to you in every way I can think of, until you get it.

RESTful Web Services covered hypermedia, but it wasn't central to the book. It was possible to skip the hypermedia parts of the book and still design a functioning API. By contrast, *RESTful Web APIs* is effectively a book about hypermedia.

I did it this way because hypermedia is the single most important aspect of REST, and the least understood. Until we all understand hypermedia, REST will continue to be viewed as a marketing buzzword rather than a serious attempt to handle the complexity of distributed computing.

What's in This Book?

The first four chapters introduce the concepts behind REST, as it applies to web APIs.

Chapter 1, Surfing the Web
: This chapter explains basic terminology using a RESTful system you're already familiar with: a website.

Chapter 2, A Simple API
: This chapter translates the lessons of the Web to a programmable API with identical functionality to the website discussed in Chapter 1.

Chapter 3, Resources and Representations
: Resources are the fundamental concept underlying HTTP, and representations are the fundamental concept underlying REST. This chapter explains how they're related.

Chapter 4, Hypermedia
: Hypermedia is the missing ingredient that ties representations together into a coherent API. This chapter shows what hypermedia is capable of, mostly using a hypermedia data format you're already familiar with: HTML.

The next four chapters describe different strategies for designing a hypermedia API:

Chapter 5, Domain-Specific Designs
: The obvious strategy is to design a completely new standard that deals with your exact problem. I use the Maze+XML standard as an example.

Chapter 6, The Collection Pattern
: One pattern in particular—the collection pattern—shows up over and over again in API design. In this chapter, I show off two different standards that capture this pattern: Collection+JSON and AtomPub.

Chapter 7, Pure-Hypermedia Designs
: When the collection pattern doesn't fit your requirements, you can convey any representation you want using a general-purpose hypermedia format. This chapter shows how it works using three general hypermedia formats (HTML, HAL, and Siren) as examples. This chapter also introduces HTML microformats and microdata, which lead in to the next chapter.

Chapter 8, Profiles
: A profile fills in the gaps between a data format (which can be used by many different APIs) and a specific API implementation. The profile format I recommend is ALPS, but I also cover XMDP and JSON-LD.

  In this chapter, my advice begins to outstrip the state of the art at the time this book was written. I had to develop the ALPS format for this book, because nothing else would do the job. If you're already familiar with hypermedia-based designs, you might be able to skip up to Chapter 8, but I don't think you should skip past it.

Chapters 9 through 13 cover practical topics like choosing the right hypermedia format and getting the most out of the HTTP protocol.

Chapter 9, The Design Procedure
: This chapter brings together everything discussed in the book so far, and gives a step-by-step guide to designing a RESTful API.

Chapter 10, The Hypermedia Zoo
: In an attempt to show what hypermedia is capable of, this chapter discusses about 20 standardized hypermedia data formats, most of them not covered elsewhere in the book.

Chapter 11, HTTP for APIs
: This chapter gives some best practices for the use of HTTP in API implementations. I also discuss some extensions to HTTP, including the forthcoming HTTP 2.0 protocol.

Chapter 12, Resource Description and Linked Data
: Linked Data is the Semantic Web community's approach to REST. JSON-LD is arguably the most important Linked Data standard. It's covered briefly in Chapter 8, and I revisit it here. This chapter also covers the RDF data model, and some RDF-based hypermedia formats that I didn't get to in Chapter 10.

Chapter 13, CoAP: REST for Embedded Systems
: This chapter closes out the core body of the book by covering CoAP, a RESTful protocol that doesn't use HTTP at all.

Appendix A, The Status Codex
: An extension of Chapter 11, this appendix provides an in-depth look at the 41 standard status codes defined in the HTTP specification, as well as a few useful codes defined as extensions.

Appendix B, The Header Codex
: Similar to Appendix A, this appendix is also an extension of Chapter 11. It provides a detailed outline of the 46 request and response headers defined in the HTTP specification, as well as a few extensions.

Appendix C, An API Designer's Guide to the Fielding Dissertation
: This appendix includes an in-depth discussion of the foundational document of REST, in terms of what it means for API design.

Glossary
: The glossary contains definitions to terms you'll frequently encounter when working with RESTful web APIs. It's a good place to turn for familiarizing yourself with basic concepts or if you need a quick, at-a-glance reminder of a particular concept's definition.

What's Not in This Book

RESTful Web Services was the first book-length treatment of REST, and it had to cover a lot of ground. Fortunately, there are now over a dozen books on various aspects of REST, and that frees up *RESTful Web APIs* to focus on the core concepts.

To keep this book focused, I've removed a few topics that you might have been expecting me to cover. I want to tell you what is not in this book, so that you don't buy it and then feel disappointed:

- Client programming is not covered here. Writing a client to consume a hypermedia-based API is a new kind of challenge. Right now, the closest thing we have to a generic API client is a library that sends HTTP requests. This was true in 2007, and it's still true. The problem is on the server side.

  When you write a client for an existing API, you're at the mercy of the API designer. I can't give you any general advice, because right now there's no consistency across APIs. That's why, in this book, I'm trying to drum up enthusiasm for a little server-side consistency. When APIs become more similar to each other, we'll be able to write more sophisticated client-side tools.

  Chapter 5 contains some sample client implementations and tries to classify different types of clients, but if you want a whole book on API clients, this is not your book. I don't think the book you want exists right now.

- The most widely deployed API client in the world is JavaScript's XMLHttpRequest library. There's a copy in every web browser, and most websites today are built atop APIs designed for consumption by XMLHttpRequest. But that's far too big a field to cover properly in this book. There are whole books written about individual JavaScript libraries.

- I spend quite a bit of time on the mechanics of HTTP (Chapter 11, Appendix A, and Appendix B), but I don't cover any given HTTP topic in a lot of depth, and there are some topics—notably HTTP intermediaries like caches and proxies—which I barely cover at all.

- *RESTful Web Services* focused heavily on breaking down your business requirements into a set of interlinked resources. My experience since 2007 has convinced me that thinking of API design as resource design is a very effective way to avoid thinking about hypermedia. This book takes a different approach, focusing on representations and state transitions rather than resources.

  That said, the resource design approach is certainly valid. For advice on moving in that direction, I recommend *RESTful Web Services Cookbook* by Subbu Allamaraju (O'Reilly).

Administrative Notes

This book has two authors (Leonard and Mike), but for the duration of this book we've merged our identities into a single authorial "I."

Nothing in this book is tied to any particular programming language. All of the code takes the form of messages (usually JSON or XML documents) sent over a network protocol (usually HTTP). I will be assuming that you're familiar with common programming concepts like antipatterns and breadth-first search, and that you have a basic understanding of how the World Wide Web works.

I won't be presenting it, but there is real code behind the servers and clients I talk about in Chapter 1, Chapter 2, and Chapter 5. You can get that code from the *RESTful Web APIs* GitHub repository, or from the official website, and run it yourself. These clients and servers are written in JavaScript, using the Node library.

I chose Node because it lets me use the same programming language for client and server code. You won't need to mentally switch back and forth between programming languages to understand both sides of a client-server transaction. Node is open source and available on Windows, Mac, and Linux systems. It is easy to install on these operating systems, and you shouldn't have much trouble getting the examples up and running.

I'm hosting the code on GitHub because that will make it easy to update the implementations over time. This also makes it possible for readers to contribute ports of the example clients and servers to other programming languages.

Understanding Standards

The World Wide Web isn't an objective thing that's out there to be studied scientifically. It's a social construct—a set of agreements to do things a certain way. Fortunately, unlike other social constructs (like etiquette), the agreements underlying the Web are generally agreed upon. The core agreements underlying the human web are RFC 2616 (the HTTP standard), the W3C's specification for HTML 4, and ECMA-262 (the standard that underlies JavaScript, also known as ECMAScript). Each standard does a different job, and over the course of this book, I'll discuss dozens of other standards designed specifically for use in APIs.

The great thing about these standards is the solid baseline they give you. You can use them to build a completely new kind of website or API, something that no one has ever tried before. Instead of having to explain your entire system to all your users, you'll only have to explain the part that's new.

The bad news is that these agreements are often borderline unreadable: long walls of ASCII text written in tooth-achingly precise English in which everyday words like "should" have technical meanings and are capitalized "SHOULD."[4] A lot of technical books are bought by people who are hoping to avoid having to read a standards document.

Well, I can't make any guarantees. If one of these standards looks like something you can use in your work, you need to be willing to dive into its spec and really understand it (or buy a book that covers it in more detail). I don't have space to give more than a basic overview of standards like Siren, CoAP, and Hydra. Not to mention that giving a

4. The meaning of "SHOULD" is given in RFC 2119.

lot of detail would bore all the readers who *don't* need those particular standards to do their work.

When navigating the forest of standards, it's useful to keep in mind that not all standards have equal force. Some are extremely well established, used by everyone, and if you go against them you're causing a lot of trouble for yourself. Other standards are just one person's opinion, and that opinion might be no better than yours.

I find it helpful to divide standards into four categories: fiat standards, personal standards, corporate standards, and open standards. I'll be using these terms throughout the book, so let me explain each one in a bit more depth before we move on.

Fiat Standards

Fiat standards aren't really standards; they're behaviors. No one agreed to them. They're just a description of the way somebody does things. The behavior may be documented, but the core assumption of a standard—that other people ought to do things the same way—is missing.

Pretty much every API today is a fiat standard, a one-off design associated with a specific company. That's why we talk about the "Twitter API," the "Facebook API," and the "Google+ API." You may need to understand these designs to do your job and you may write your own clients for these designs, but unless you work for the company in question, there's no expectation that you should use this design for *your* API. If you reuse a fiat standard, we don't say your API conforms to a standard; we say it's a clone.

The main problem I'm trying to solve in this book is that hundreds of person-years of design work is locked up in fiat standards where it can't be reused. This needs to stop. Designing a new API today means reinventing a long series of wheels. Once your API is finished, your client developers have to reinvent corresponding wheels on the client side.

Even under ideal circumstances, your API will be a fiat standard, since your business requirements will be slightly different from everyone else's. But ideally a fiat standard would be just a light gloss over a number of other standards.

When I describe a fiat standard, I'll link to its human-readable documentation.

Personal Standards

Personal standards are standards—you're invited to read the documents and implement the standards yourself—but they're just one person's opinion. The Maze+XML standard I describe in Chapter 5 is a good example. There's no expectation that Maze+XML is the standard way to implement a maze game API, but if it works for you, you might as well use it. Someone else has done the design work for you.

Personal standards generally use less formal language than other kinds of standards. Many open standards start off as personal standards—as side projects that are formalized after a lot of experimentation. Siren, which I cover in Chapter 7, is a good example.

When I describe a personal standard, I'll link to its specification.

Corporate Standards

Corporate standards are created by a consortium of companies trying to solve a problem that plagues them all, or by a single company trying to solve a recurring problem on behalf of its customers. Corporate standards tend to be better defined and to use more formal language than personal standards, but they have no more force than personal standards. They're just one company's (or a group of companies') opinion.

Corporate standards include Activity Streams and schema.org's microdata schemas, both of which are covered in Chapter 10. Many industry standards start off as corporate standards. OData (also discussed in Chapter 10) started as a Microsoft project, but it was submitted to OASIS in 2012 and will eventually become an OASIS standard.

When I describe a corporate standard, I'll link to its specification.

Open Standards

An open standard has gone through a process of design by committee, or at least had an open comment period during which a lot of people read the specification, complained about it, and made suggestions for improvement. At the end of this process, the specification was blessed by some kind of recognized standards body.

This process gives an open standard a certain amount of moral force. If there's an open standard that does more or less what you want, you really should use it instead of making up your own fiat standard. The design process and the comment period probably turned up a lot of issues that you won't encounter until it's too late.

In general, open standards come with some kind of agreement that promises you can implement them without getting hit with a patent infringement lawsuit from a company that was involved in the standards process. By contrast, implementing someone else's fiat standard may *incite* them to file a patent infringement lawsuit against you.

A few open standards mentioned in this book came out of the big-name standards bodies: ANSI, ECMA, ISO, OASIS, and especially the W3C. I can't say what it's like to sit on one of these standards bodies, because I've never done it. But the most important standards body[5] is one anyone can contribute to: the IETF, the group that manages the all-important RFCs.

5. For the purposes of this book, anyway. If you need standard sizes for screws and bolts, you want ANSI or ISO.

Requests for Comments (RFCs) and Internet-Drafts

Most RFCs are created through a process called the Standards Track. Throughout this book, I'll be referencing documents that are in different places on the Standards Track. I'd like to briefly discuss how the track works, so that you'll know how seriously to take my recommendations.

An RFC begins life as an *Internet-Draft*. This is a document that looks like a standards document, but you're not supposed to build implementations based on it. You're supposed to find problems with the specification and give feedback.

An Internet-Draft has a built-in lifetime of six months. Six months after it is published, a draft must be approved as an RFC or replaced with an updated draft. If neither of those things happens, then the draft expires and should not be used for anything. On the other hand, if the draft is approved, it expires immediately and is replaced by an RFC.

Because of the built-in expiration date, and because an Internet-Draft isn't technically any kind of standard, it's tricky business mentioning them in a book. At the same time, API design is a field that's changing rapidly, and an Internet-Draft is better than nothing. I will be mentioning many Internet-Drafts in this book under the assumption that they'll become RFCs without major changes. That assumption has held up pretty well; several Internet-Drafts that I mention here became RFCs while I was writing the book. If a particular draft doesn't pan out, all I can do is apologize in advance.

RFCs and Internet-Drafts are given code names. When I describe one of these, I *won't* link to its specification. I'll just refer to it by its code and let you look it up. For example, I'll refer to the HTTP/ 1.1 specification as RFC 2616. I'll refer to an Internet-Draft by its name. For example, I'll use "draft-snell-link-method" to refer to the proposal to add LINK and UNLINK methods to HTTP.

Whenever you see one of these code names, you can do a web search and find the latest version of the RFC or Internet-Draft. If an Internet-Draft becomes an RFC after this book is published, the final version of the Internet-Draft will link to the RFC.

When I describe a W3C or OASIS standard, I'll link to the specification, because those standards aren't given code names.

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, if this book includes code examples, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from 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 APIs* by Leonard Richardson and Mike Amundsen (O'Reilly). Copyright 2013 Leonard Richardson and amundsen.com, Inc., and Sam Ruby. 978-1-449-35806-8."

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

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

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://oreil.ly/RESTful-Web-APIs*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We owe a debt of thanks to Glenn Block who spent untold hours listening to ideas and working through real code to test those ideas. To Benjamin Young and all the folks at RESTFest who agreed to be part of our experiments, and who gave great feedback and advice even when we didn't want to hear it. To Mike's colleagues at Layer 7 Technologies, including Dimitri Sirota and Matt McLarty, who supported and encouraged his work on this project. To Sam Ruby and Mike Loukides, who were essential to *RESTful Web Services*, this book's predecessor. To Sumana Harihareswara, Leonard's supportive wife. To the social communities that create an excellent place to collaborate and converse on REST and APIs; especially Yahoo's REST-Discuss, Google Groups' API-Craft, and the Hypermedia group at LibreList.

And finally, to all those who read the early drafts of this manuscript and provided much-needed criticism and support: Carsten Bormann, Todd Brackley, Tom Christie, Timothy Haas, Jamie Hodge, Alex James, David Jones, Markus Lanthaler, Even Maler, Mark Nottingham, Cheryl Phair, Sergey Shishkin, Brian Sletten, Mark Stafford, Stefan Tilkov, Denny Vrandečić, Ruben Verborgh, and Andrew Wahbe.

CHAPTER 1
Surfing the Web

The World Wide Web became popular because ordinary people can use it to do really useful things with minimal training. But behind the scenes, the Web is also a powerful platform for distributed computing.

The principles that make the Web usable by ordinary people also work when the "user" is an automated software agent. A piece of software designed to transfer money between bank accounts (or carry out any other real-world task) can accomplish the task using the same basic technologies a human being would use.

As far as this book is concerned, the Web is based on three technologies: the URL naming convention, the HTTP protocol, and the HTML document format. URL and HTTP are simple, but to apply them to distributed programming you must understand them in more detail than the average web developer does. The first few chapters of this book are dedicated to giving you this understanding.

The story of HTML is a little more complicated. In the world of web APIs, there are dozens of data formats competing to take the place of HTML. An exploration of these formats will take up several chapters of this book, starting in Chapter 5. For now, I want to focus on URL and HTTP, and use HTML solely as an example.

I'm going to start off by telling a simple story about the World Wide Web, as a way of explaining the principles behind its design and the reasons for its success. The story needs to be simple because although you're certainly familiar with the Web, you might not have heard of the concepts that make it work. I want you to have a simple, concrete example to fall back on if you ever get confused about terminology like "hypermedia as the engine of application state."

Let's get started.

Episode 1: The Billboard

One day Alice is walking around town and she sees a billboard (Figure 1-1).

Figure 1-1. The billboard

(By the way, this fictional billboard advertises a real website that I designed for this book. You can try it out yourself.)

Alice is old enough to remember the mid-1990s, so she recalls the public's reaction when URLs started showing up on billboards. At first, people made fun of these weird-looking strings. It wasn't clear what "http://" or "youtypeitwepostit.com" meant. But 20 years later, everyone knows what to do with a URL: you type it into the address bar of your web browser and hit Enter.

And that's what Alice does: she pulls out her mobile phone and puts *http://www.youtypeitwepostit.com/* in her browser's address bar. The first episode of our story ends on a cliffhanger: what's at the other end of that URL?

Resources and Representations

Sorry for interrupting the story, but I need to introduce some basic terminology. Alice's web browser is about to send an HTTP request to a web server—specifically, to the URL *http://www.youtypeitwepostit.com/*. One web server may host many different URLs, and each URL grants access to a different bit of the data on the server.

We say that a URL is the URL of some thing: a product, a user, the home page. The technical term for the thing named by a URL is *resource*.

The URL *http://www.youtypeitwepostit.com/* identifies a resource—probably the home page of the website advertised on the billboard. But you won't know for sure until we resume the story and Alice's web browser sends the HTTP request.

When a web browser sends an HTTP request for a resource, the server sends a document in response (usually an HTML document, but sometimes a binary image or something else). Whatever document the server sends, we call that document a *representation* of the resource.

So each URL identifies a resource. When a client makes an HTTP request to a URL, it gets a representation of the underlying resource. The client never sees a resource directly.

I'll talk a lot more about resources and representations in Chapter 3. Right now I just want to use the terms resource and representation to discuss the principle of addressability, to which I'll now turn.

Addressability

A URL identifies one and only one resource. If a website has two conceptually different things on it, we expect the site to treat them as two resources with different URLs. We get frustrated when a website violates this rule. Websites for restaurants are especially bad about this. Frequently, the whole site is buried inside a Flash interface and there's no URL that points to the menu or to the map that shows where the restaurant is located —things we would like to talk about on their own.

The principle of addressability just says that every resource should have its own URL. If something is important to your application, it should have a unique name, a URL, so that you and your users can refer to it unambiguously.

Episode 2: The Home Page

Back to our story. When Alice enters the URL from the billboard into her browser's address bar, it sends an HTTP request over the Internet to the web server at *http://www.youtypeitwepostit.com/*:

```
GET / HTTP/1.1
Host: www.youtypeitwepostit.com
```

The web server handles this request (neither Alice nor her web browser need to know how) and sends a response:

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <div>
          <h1>You type it, we post it!</h1>
```

```
            <p>Exciting! Amazing!</p>

            <p class="links">
              <a href="/messages">Get started</a>
              <a href="/about">About this site</a>
            </p>
          </div>
        </body>
    </html>
```

The 200 at the beginning of the response is a *status code*, also called a *response code*. It's a quick way for the server to tell the client approximately what happened to the client's request. There are a lot of HTTP status codes, and I cover them all in Appendix A, but the most common one is the one you see here. 200 (`OK`) means that the request was fulfilled with no problems.

Alice's web browser decodes the response as an HTML document and displays it graphically (see Figure 1-2).

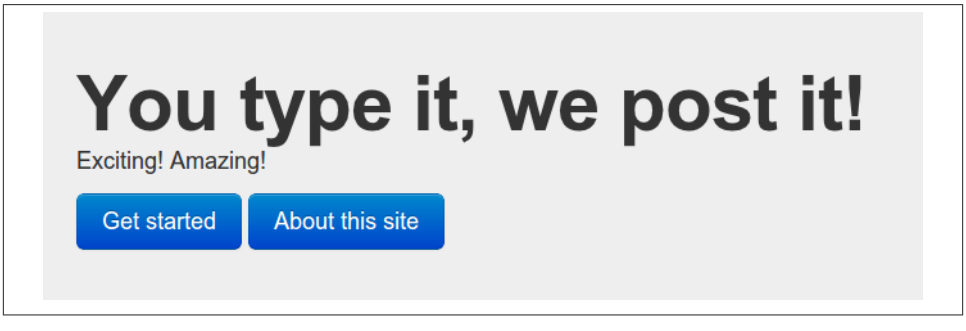

Figure 1-2. You Type It… home page

Now Alice can read the web page and understand what the billboard was talking about. It was advertising a microblogging site, similar to Twitter. Not as exciting as advertised on the billboard, but good enough as an example.

Alice's first real interaction with the web server reveals a couple more important features of the Web.

Short Sessions

At this point in the story, Alice's web browser is displaying the site's home page. From her perspective, she's "landed" on that page, which is is her current "location" in cyberspace. But as far as the server is concerned, Alice isn't anywhere. The server has already forgotten about her.

HTTP sessions last for one request. The client sends a request, and the server responds. This means Alice could turn her phone off overnight, and when her browser restored the page from its internal cache, she could click on one of the two links on this page and it would still work. (Compare this to an SSH session, which is terminated if you turn your computer off.)

Alice could leave this web page open in her phone for six months, and when she finally clicks on a link, the web server would respond as if she'd only waited a few seconds. The web server isn't sitting up late at night worrying about Alice. When she's not making an HTTP request, the server doesn't know Alice exists.

This principle is sometimes called statelessness. I think this is a confusing term because the client and the server in this system both keep state; they just keep different *kinds* of state. The term "statelessness" is getting at the fact that the *server* doesn't care what state the *client* is in. (I'll talk more about the different kinds of state in the following sections.)

Self-Descriptive Messages

It's clear from looking at the HTML that this site is more than just a home page. The markup for the home page contains two links: one to the relative URL */about* (i.e., to *http://www.youtypeitwepostit.com/about*) and one to */messages* (i.e., *http://www.youtypeitwepostit.com/messages*). At first Alice only knew one URL—the URL to the home page—but now she knows three. The server is slowly revealing its structure to her.

We can draw a map of the website so far (Figure 1-3), as revealed to Alice by the server.

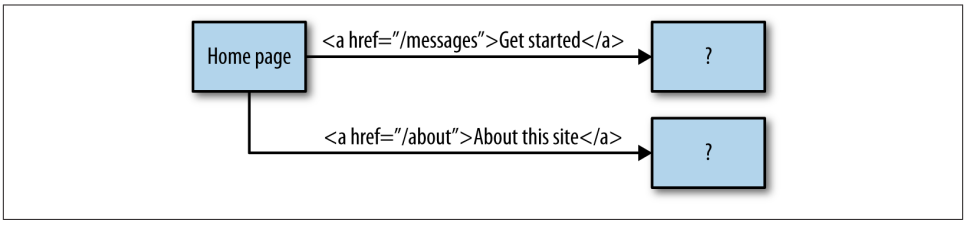

Figure 1-3. A map of the website

What's on the other end of the */messages* and */about* links? The only way to be sure is to follow them and find out. But Alice can look at the HTML markup, or her browser's graphical rendering of the markup, and make an educated guess. The link with the text "About this site" probably goes to a page talking about the site. That's nice, but the link with the text "Get started" is probably the one that gets her closer to actually posting a message.

When you request a web page, the HTML document you receive doesn't just give you the immediate information you asked for. The document also helps you answer the question of what to do next.

Episode 3: The Link

After reading the home page, Alice decides to give this site a try. She clicks the link that says "Get started." Of course, whenever you click a link in your web browser, you're telling your web browser to make an HTTP request.

The code for the link Alice clicked on looks like this:

```
<a href="/messages">Get started</a>
```

So her browser makes this HTTP request to the same server as before:

```
GET /messages HTTP/1.1
Host: www.youtypeitwepostit.com
```

That GET in the request is an *HTTP method*, also known as an *HTTP verb*. The HTTP method is the client's way of telling the server what it wants to do to a resource. "GET" is the most common HTTP method. It means "give me a representation of this resource." For a web browser, GET is the default. When you follow a link or type a URL into the address bar, your browser sends a GET request.

The server handles this particular GET request by sending a representation of */messages*:

```
HTTP/1.1 200 OK
Content-type: text/html
...

<!DOCTYPE html>
<html>
    <head>
        <title>Messages</title>
    </head>
    <body>
      <div>
        <h1>Messages</h1>

        <p>
          Enter your message below:
        </p>

        <form action="http://youtypeitwepostit.com/messages" method="post">
            <input type="text" name="message" value="" required="true"
                   maxlength="6"/>
            <input type="submit" value="Post" />
        </form>

        <div>
```

```
            <p>
              Here are some other messages, too:
            </p>
            <ul>
              <li><a href="/messages/32740753167308867">Later</a></li>
              <li><a href="/messages/7534227794967592">Hello</a></li>
            </ul>
          </div>

          <p class="links">
            <a href="http://youtypeitwepostit.com/">Home</a>
          </p>

        </div>
      </body>
    </html>
```

As before, Alice's browser renders the HTML graphically (Figure 1-4).

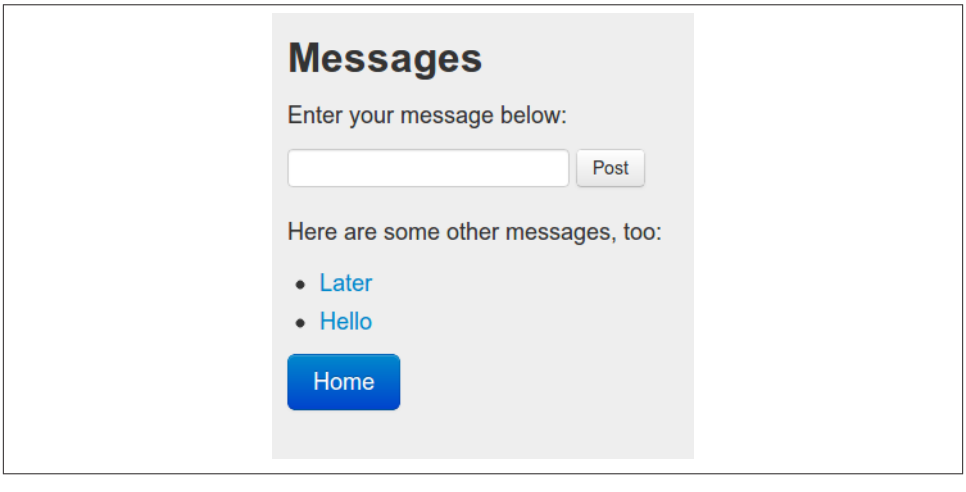

Figure 1-4. You Type It… "Get started" page

When Alice looks at the graphical rendering, she sees that this page is a list of messages other people have published on the site. Right at the top there's an inviting text box and a Post button.

Now we've revealed a little more about how the server works. Figure 1-5 shows an updated map of the site, as seen by Alice's browser.

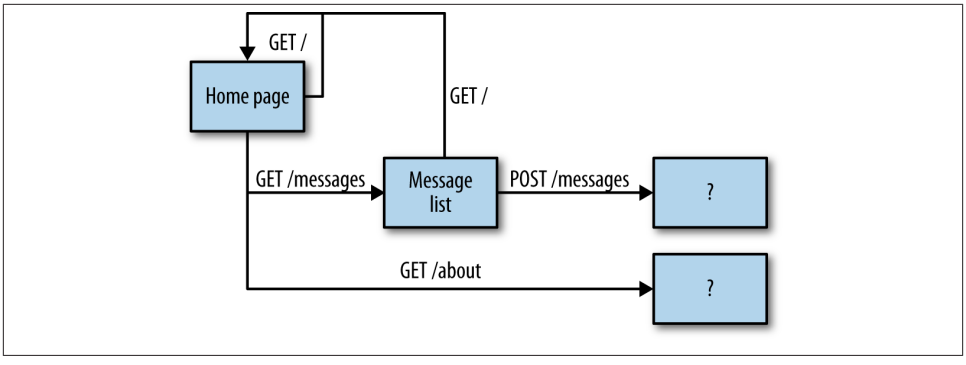

Figure 1-5. The browser's view of You Type It…

Standardized Methods

Both of Alice's HTTP requests used GET as their HTTP method. But there's a bit of HTML in the latest representation that will trigger an HTTP POST request if Alice clicks the Post button:

```
<form action="http://youtypeitwepostit.com/messages" method="post">
    <input type="text" name="message" value="" required="true"
           maxlength="6"/>
    <input type="submit" />
</form>
```

The HTTP standard (RFC 2616) defines eight methods a client can apply to a resource. In this book, I'll focus on five of them: GET, HEAD, POST, PUT, and DELETE. In Chapter 3, I'll cover these methods in detail, along with an extension method, PATCH, designed specifically for use in web APIs. Right now the important thing to keep in mind is that there are a small number of standard methods.

It's not impossible to come up with a new HTTP method (it happened with PATCH), but it's a very big deal. This is not like a programming language, where you can name your methods whatever you want. When I built the simple microblogging website for use in this example, I didn't define new HTTP methods like GETHOMEPAGE and HELLOPLEASESHOWMETHEMESSAGELISTTHANKSBYE. I used GET for both "show the home page" and "show the message list," because in both cases GET ("give me a representation of this resource") was the best match between HTTP's interface and what I wanted to do. I distinguished between the home page and the message list not by defining new methods, but by treating those two documents as separate resources, each with its own URL, each accessible through GET.

Episode 4: The Form and the Redirect

Back to our story. Alice is tempted by the form on the microblogging site. She types in "Test" and clicks the Post button.:

Again, Alice's browser makes an HTTP request:

```
POST /messages HTTP/1.1
Host: www.youtypeitwepostit.com
Content-type: application/x-www-form-urlencoded

message=Test&submit=Post
```

And the server responds with the following:

```
HTTP/1.1 303 See Other
Content-type: text/html
Location: http://www.youtypeitwepostit.com/messages/5266722824890167
```

When Alice's browser made its two GET requests, the server sent the HTTP status code 200 ("OK") and provided an HTML document for Alice's browser to render. There's no HTML document here, but the server did provide a link to another URL, in the `Location` header—and here, the status code at the beginning of the response is 303 ("See Other"), not 200 ("OK").

Status code 303 tells Alice's browser to *automatically* make a fourth HTTP request, to the URL given in the `Location` header. Without asking Alice's permission, her browser does just that:

```
GET /messages/5266722824890167 HTTP/1.1
```

This time, the server responds with 200 ("OK") and an HTML document:

```
HTTP/1.1 200 OK
Content-type: text/html

<!DOCTYPE html>
<html>
    <head>
        <title>Message</title>
    </head>
    <body>
        <div>
            <h2>Message</h2>
            <dl>
                <dt>ID</dt><dd>2181852539069950</dd>
                <dt>DATE</dt><dd>2014-03-28T21:51:08Z</dd>
                <dt>MSG</dt><dd>Test</dd>
            </dl>
            <p class="links">
                <a href="http://www.youtypeitwepostit.com/">Home</a>
            </p>
```

```
        </div>
    </body>
</html>
```

Alice's browser displays this document graphically (Figure 1-6), and, finally, goes back to waiting for Alice's input.

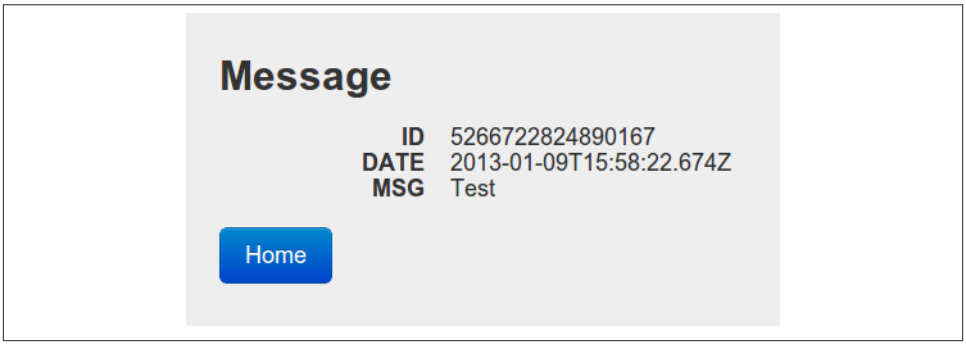

Figure 1-6. You Type It… posted message

I'm sure you've encountered HTTP redirects before, but HTTP is full of small features like this, and some may be new to you. There are many ways for the server to tell the client to handle a response differently, and ways for the client to attach conditions or extra features to its request. A big part of API design is the proper use of these features. Chapter 11 covers the features of HTTP that are most important to web APIs, and Appendix A and Appendix B provide supplementary information on this topic.

By looking at the graphical rendering, Alice sees that her message ("Test") is now a fully fledged post on YouTypeItWePostIt.com. Our story ends here—Alice has accomplished her goal of trying out the microblogging site. But there's a lot to be learned from these four simple interactions.

Application State

Figure 1-7 is a state diagram that shows Alice's entire adventure from the perspective of her web browser.