Web Services for the Real World

RESTful Web Services

O'REILLY®

Leonard Richardson & Sam Ruby

Foreword by David Heinemeier Hansson

RESTful Web Services
by Leonard Richardson and Sam Ruby

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

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]

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

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.

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.

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`	`200 OK`
`Host: www.example.com`	`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.

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.

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.

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 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/*).

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

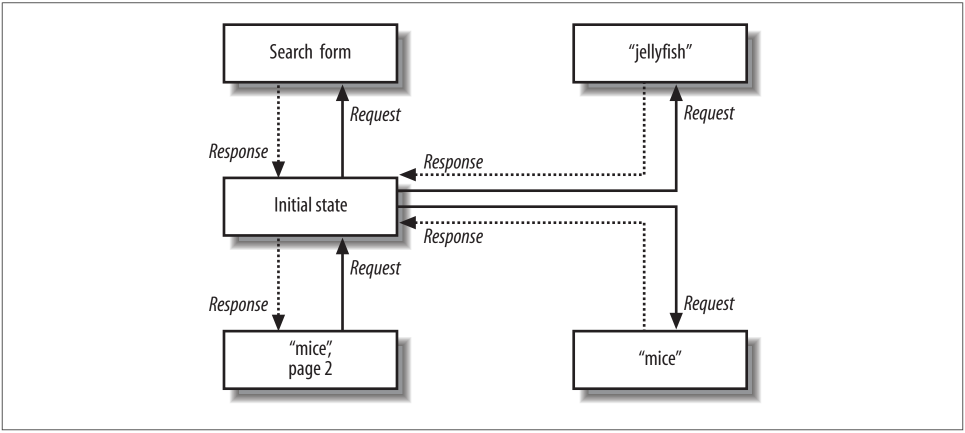


Figure 4-1. A stateless search engine

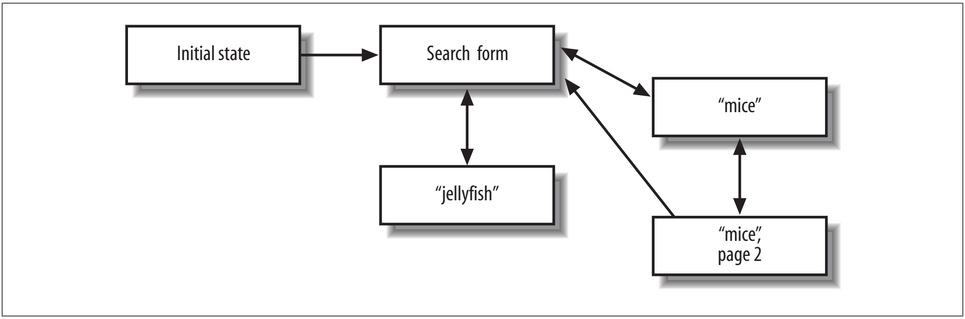


Figure 4-2. A stateful search engine

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 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 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 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.

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.

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.

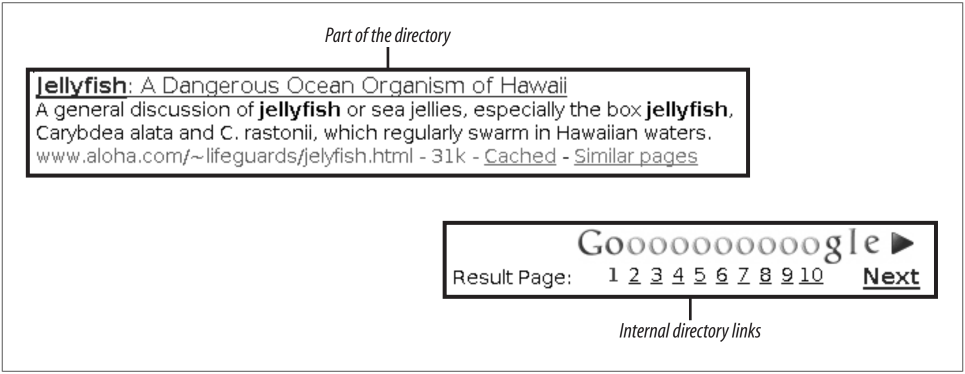
It's RESTful to keep this information in the HTTP headers, and it's RESTful to put it in the URI. I recommend keeping as much of this information as possible in the URI, and as little as possible in request metadata. I think URIs are more useful than metadata. URIs get passed around from person to person,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



Jellyfish: A Dangerous Ocean Organism of Hawaii
A general discussion of **jellyfish** or sea jellies, especially the box **jellyfish**, Carybdea alata and C. rastonii, which regularly swarm in Hawaiian waters.
www.aloha.com/~lifeguards/jelyfish.html - 31k - Cached - Similar pages

Goooooooooogle ▶
Result Page: 1 2 3 4 5 6 7 8 9 10 **Next**

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.

- 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.

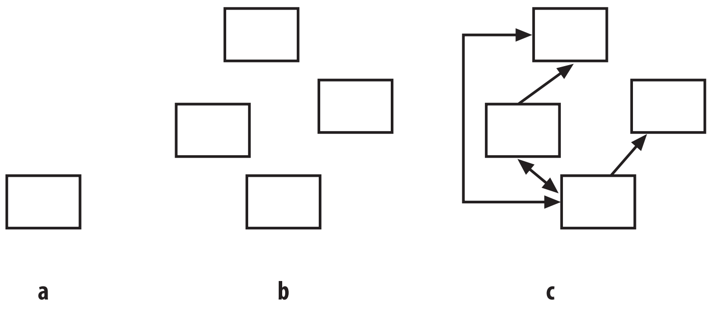
Earlier in this chapter, I showed what might happen if HTTP was a stateful protocol like FTP. Figure 4-2 shows the paths a stateful HTTP client might take during a “session” with `www.google.com`. HTTP doesn’t really work that way, but that figure does a good job of showing how we use the human web. 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.



All three services expose the same functionality, but their usability increases toward the right.

•Service A is a typical RPC-style service, exposing everything through a single URI. It's neither addressable nor connected.

•Service B is addressable but not connected: there are no indications of the relationships between resources. This might be a REST-RPC hybrid service, or a RESTful service like Amazon S3.

•Service C is addressable and well-connected: resources are linked to each other in ways that (presumably) make sense. This could be a fully RESTful service.

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>
```

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.

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.

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 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 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.

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.

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 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 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.