**Learn REST: A Tutorial**

A fast-training course for **REST** - *Re*presentational *S*tate *T*ransfer, a new approach to systems architecture and a lightweight alternative to web services

[1. What is REST?](http://rest.elkstein.org/2008/02/what-is-rest.html)

REST stands for **Re**presentational **S**tate **T**ransfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used.

REST is *an architecture style* for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines.

* In many ways, the World Wide Web itself, based on HTTP, can be viewed as a REST-based architecture.

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, et al.). Later, we will see how much more simple REST is.

* Despite being simple, REST is fully-featured; there's basically nothing you can do in Web Services that can't be done with a RESTful architecture.

REST is not a "standard". There will never be a W3C recommendataion for REST, for example. And while there are REST programming frameworks, working with REST is so simple that you can often "roll your own" with standard library features in languages like Perl, Java, or C#.

BY DR. M. ELKSTEIN [47 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=7197089526757106233)

[2. REST as Lightweight Web Services](http://rest.elkstein.org/2008/02/rest-as-lightweight-web-services.html)

As a programming approach, REST is a lightweight alternative to Web Services and RPC.

Much like Web Services, a REST service is:

* Platform-independent (you don't care if the server is Unix, the client is a Mac, or anything else),
* Language-independent (C# can talk to Java, etc.),
* Standards-based (runs on top of HTTP), and
* Can easily be used in the presence of firewalls.

Like Web Services, REST offers no built-in security features, encryption, session management, QoS guarantees, etc. But also as with Web Services, these can be added by building on top of HTTP:

* For security, username/password tokens are often used.
* For encryption, REST can be used on top of HTTPS (secure sockets).
* ... etc.

One thing that is *not* part of a good REST design is cookies: The "ST" in "RE**ST**" stands for "State Transfer", and indeed, in a good REST design operations are self-contained, and each request carries with it (transfers) all the information (state) that the server needs in order to complete it.

BY DR. M. ELKSTEIN [10 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=4660893850404656480)

[3. How Simple is REST?](http://rest.elkstein.org/2008/02/how-simple-is-rest.html)

Let's take a simple web service as an example: querying a phonebook application for the details of a given user. All we have is the user's ID.

Using Web Services and SOAP, the request would look something like this:

<?xml version="1.0"?>

<soap:Envelope

xmlns:soap="http://www.w3.org/2001/12/soap-envelope"

soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:body pb="http://www.acme.com/phonebook">

<pb:GetUserDetails>

<pb:UserID>12345</pb:UserID>

</pb:GetUserDetails>

</soap:Body>

</soap:Envelope>

(The details are not important; this is just an example.) The entire shebang now has to be sent (using an HTTP POST request) to the server. The result is probably an XML file, but it will be embedded, as the "payload", inside a SOAP response envelope.

And with REST? The query will probably look like this:

http://www.acme.com/phonebook/UserDetails/12345

Note that this isn't the request body -- it's just a URL. This URL is sent to the server using a simpler GET request, and the HTTP reply is the raw result data -- not embedded inside anything, just the data you need in a way you can directly use.

* It's easy to see why Web Services are often used with libraries that create the SOAP/HTTP request and send it over, and then parse the SOAP response.
* With REST, a simple network connection is all you need. You can even test the API directly, using your browser.
* Still, REST libraries (for simplifying things) do exist, and we will discuss some of these later.

Note how the URL's "method" part is not called "GetUserDetails", but simply "UserDetails". It is a common convention in REST design to use *nouns* rather than *verbs* to denote simple *resources*.

The letter analogy  
A nice analogy for REST vs. SOAP is mailing a letter: with SOAP, you're using an envelope; with REST, it's a postcard. Postcards are easier to handle (by the receiver), waste less paper (i.e., consume less bandwidth), and have a short content. (Of course, REST requests aren't really limited in length, esp. if they use POST rather than GET.)

But don't carry the analogy too far: unlike letters-vs.-postcards, REST is every bit as secure as SOAP. In particular, REST can be carried over secure sockets (using the HTTPS protocol), and content can be encrypted using any mechanism you see fit. Without encryption, REST and SOAP are both insecure; with proper encryption in place, both are equally secure.

BY DR. M. ELKSTEIN [21 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=1127899704173401350)

[4. More Complex REST Requests](http://rest.elkstein.org/2008/02/more-complex-rest-requests.html)

The previous section included a simple example for a REST request -- with a single parameter.

REST can easily handle more complex requests, including multiple parameters. In most cases, you'll just use HTTP GET parameters in the URL.

For example:

http://www.acme.com/phonebook/UserDetails?firstName=John&lastName=Doe

If you need to pass long parameters, or binary ones, you'd normally use HTTP POST requests, and include the parameters in the POST body.

As a rule, GET requests should be for read-only queries; they should not change the state of the server and its data. For creation, updating, and deleting data, use POST requests. (POST can also be used for read-only queries, as noted above, when complex parameters are required.)

* In a way, this web page (like most others) can be viewed as offering services via a REST API; you use a GET request to read data, and a POST request to post a comment -- where more and longer parameters are required.

While REST services might use XML in their *responses* (as one way of organizing structured data), REST *requests* rarely use XML. As shown above, in most cases, request parameters are simple, and there is no need for the overhead of XML.

* One advantage of using XML is type safety. However, in a stateless system like REST, you should *always* verify the validity of your input, XML or otherwise!

[5. REST Server Responses](http://rest.elkstein.org/2008/02/rest-server-responses.html)

A server response in REST is often an XML file; for example,

<parts-list>

<part id="3322">

<name>ACME Boomerang</name>

<desc>

Used by Coyote in <i>Zoom at the Top</i>, 1962

</desc>

<price currency="usd" quantity="1">17.32</price>

<uri>http://www.acme.com/parts/3322</uri>

</part>

<part id="783">

<name>ACME Dehydrated Boulders</name>

<desc>

Used by Coyote in <i>Scrambled Aches</i>, 1957

</desc>

<price currency="usd" quantity="pack">19.95</price>

<uri>http://www.acme.com/parts/783</uri>

</part>

</parts-list>

However, other formats can also be used; unlike SOAP services, REST is *not* bound to XML in any way. Possible formats include [CSV](http://en.wikipedia.org/wiki/Comma-separated_values) (comma-separated values) and [JSON](http://en.wikipedia.org/wiki/JSON) (JavaScript Object Notation).

Each format has its own advantages and disadvantages. XML is easy to expand (clients should ignore unfamiliar fields) and is type-safe; CSV is more compact; and JSON is trivial to parse in JavaScript clients (and easy to parse in other languages, too).

One option is *not* acceptable as a REST response format, except in very specific cases: HTML, or any other format which is meant for human consumption and is not easily processed by clients. The specific exception is, of course, when the REST service is documented to return a human-readable document; and when viewing the entire WWW as a RESTful application, we find that HTML is in fact the most common REST response format...

BY DR. M. ELKSTEIN [18 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=7769993720569953596)

[6. Real REST Examples](http://rest.elkstein.org/2008/02/real-rest-examples.html)

Here's a very partial list of service providers that use a REST API. Note that some of them also support a WSDL (Web Services) API, in addition, so you can pick which to use; but in most cases, when both alternatives are available, REST calls are easier to create, the results are easier to parse and use, and it's also less resource-heavy on your system.  
So without further ado, some REST services:

* The Google Glass API, known as "Mirror API", is a pure REST API. Here is [an excellent video talk](http://www.youtube.com/watch?feature=player_embedded&v=JpWmGX55a40) about this API. (The actual API discussion starts after 16 minutes or so.)
* Twitter has a [**REST API**](https://dev.twitter.com/docs/api) (in fact, this was their original API and, so far as I can tell, it's still the main API used by Twitter application developers),
* [**Flickr**](http://www.flickr.com/services/api/),
* **Amazon.com** offer several REST services, e.g., for their [S3 storage solution](http://docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html),
* [**Atom**](http://en.wikipedia.org/wiki/Atom_(standard)) is a RESTful alternative to RSS,
* [**Tesla Model S**](http://docs.timdorr.apiary.io/) uses an (undocumented) REST API between the car systems and its Android/iOS apps.

(This is far from an exhaustive list.)

BY DR. M. ELKSTEIN [24 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=2877263407759349042)

[7. AJAX and REST](http://rest.elkstein.org/2008/02/ajax-and-rest.html)

[AJAX](http://en.wikipedia.org/wiki/AJAX) is a popular web development technique that makes web pages interactive using JavaScript.

In AJAX, requests are sent to the server using XMLHttpRequest objects. The response is used by the JavaScript code to dynamically change the current page.

In many ways, AJAX applications follow the REST design principles. Each XMLHttpRequestcan be viewed as a REST service request, sent using GET. And the response is often in JSON, a popular response format for REST. (See [REST Server Responses](http://rest.elkstein.org/2008/02/rest-server-responses.html), above.)

To make your AJAX application truly RESTful, follow the standard REST design principles ([discussed later](http://rest.elkstein.org/2008/02/rest-design-guidelines.html)). You will find that most of them contribute to a good design, even if you don't think of your architecture in terms of REST.

A later section provides code samples for [issuing HTTP requests in JavaScript](http://rest.elkstein.org/2008/02/using-rest-in-javascript.html), but if you've done any AJAX programming, you are already familiar with all that.

BY DR. M. ELKSTEIN [1 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=426453086212387048)

[8. REST Architecture Components](http://rest.elkstein.org/2008/02/rest-architecture-components.html)

Key components of a REST architecture:

* **Resources**, which are identified by logical URLs. Both *state* and *functionality* are represented using resources.
  + The logical URLs imply that the resources are *universally addressable* by other parts of the system.
  + **Resources are the key element of a true RESTful design**, as opposed to "methods" or "services" used in RPC and SOAP Web Services, respectively. You do not issue a "getProductName" and then a "getProductPrice" RPC calls in REST; rather, you view the product data as a resource -- and this resource should contain all the required information (or links to it).
* **A web of resources**, meaning that a single resource should not be overwhelmingly large and contain too fine-grained details. Whenever relevant, a resource should contain *links* to additional information -- just as in web pages.
* The system has a **client-server**, but of course one component's server can be another component's client.
* There is **no connection state**; interaction is stateless (although the servers and resources can of course be stateful). Each new request should carry all the information required to complete it, and must not rely on previous interactions with the same client.
* Resources should be **cachable** whenever possible (with an expiration date/time). The protocol must allow the server to explicitly specify which resources may be cached, and for how long.
  + Since HTTP is universally used as the REST protocol, the HTTP cache-control headers are used for this purpose.
  + Clients must respect the server's cache specification for each resource.
* **Proxy servers** can be used as part of the architecture, to improve performance and scalability. Any standard HTTP proxy can be used.

Note that your application can *use REST services* (as a client) without being a REST architecture by itself; e.g., a single-machine, non-REST program can access 3rd-party REST services.

[REST Design Guidelines](http://rest.elkstein.org/2008/02/rest-design-guidelines.html)

Some soft guidelines for designing a REST architecture:

1. Do not use "physical" URLs. A physical URL points at something physical -- e.g., an XML file: "http://www.acme.com/inventory/product003.xml". A *logical* URL does not imply a physical file: "http://www.acme.com/inventory/product/003".
   * Sure, even with the .xml extension, the content could be dynamically generated. But it should be "humanly visible" that the URL is logical and not physical.
2. Queries should not return an overload of data. If needed, provide a paging mechanism. For example, a "product list" GET request should return the first *n*products (e.g., the first 10), with next/prev links.
3. Even though the REST response can be anything, make sure it's well documented, and do not change the output format lightly (since it will break existing clients).
   * Remember, even if the output is human-readable, your clients aren't human users.
   * If the output is in XML, make sure you document it with a schema or a DTD.
4. Rather than letting clients construct URLs for additional actions, include the actual URLs with REST responses. For example, a "product list" request could return an ID per product, and the specification says that you should use http://www.acme.com/product/*PRODUCT\_ID* to get additional details. That's bad design. Rather, the response should include the actual URL with each item: http://www.acme.com/product/001263, etc.
   * Yes, this means that the output is larger. But it also means that you can easily direct clients to new URLs as needed, without requiring a change in client code.
5. GET access requests should never cause a state change. Anything that changes the server state should be a POST request (or other HTTP verbs, such as DELETE).

BY DR. M. ELKSTEIN [19 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=9012084198722981760)

[10. ROA vs. SOA, REST vs. SOAP](http://rest.elkstein.org/2008/02/roa-vs-soa-rest-vs-soap.html)

ROA (REST Oriented Architecture) is just a fancy name for a SOA (Service Based Architecture) using REST services.

The main advantage of SOAP-based SOA over ROA is the more mature tool support; however, this could change over time. Another SOA advantages include the type-safety of XML requests (for responses, ROA can also use XML if the developers desire it).

The main advantage of ROA is ease of implementation, agility of the design, and the lightweight approach to things. In a way, SOA and SOAP is for people in business suits; that's what you'll find used in the banking and finance industries. Conversely, somebody that needs something up-and-running quickly, with good performance and low overhead, is often better off using REST and ROA.

For example, when explaining why they chose REST over SOAP, Yahoo! people write that they "*believe REST has a lower barrier to entry, is easier to use than SOAP, and is entirely sufficient for [Yahoo's] services*" ([Yahoo! Developer Network FAQ](http://developer.yahoo.com/faq/), as of February 2008). This is true not only of REST vs. SOAP but also of ROA vs. SOA in general.

Another advantage of REST lies with performance: with better cache support, lightweight requests and responses, and easier response parsing, REST allows for nimbler clients and servers, and reduces network traffic, too.

As REST matures, expect it to become better understood and more popular even in more conservative industries.

**A few words on hype vs. reality.** In the comments below, I've linked to Pete Lacey's excellent criticism of SOAP, "[The S Stands for Simple](http://wanderingbarque.com/nonintersecting/2006/11/15/the-s-stands-for-simple/)". And here's another one, by Alex Bell, published in *Communications of the ACM* (vol. 51, no. 10, October '08): "[DOA with SOA](http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=464)".

BY DR. M. ELKSTEIN [13 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=5738513359339784109)

[11. Documenting REST Services: WSDL and WADL](http://rest.elkstein.org/2008/02/documenting-rest-services-wsdl-and-wadl.html)

**WSDL**, a W3C recommendation, is the Web Services Description Language. It is commonly used to spell out in detail the services offered by a SOAP server. While WSDL is flexible in service binding options (for example, services can be offered via SMTP mail servers), it did not originally support HTTP operations other than GET and POST. Since REST services often use other HTTP verbs, such as PUT and DELETE, WSDL was a poor choice for documenting REST services.

With [version 2.0](http://rest.elkstein.org/search?updated-max=2008-02-12T18:00:00-08:00&max-results=4&start=4&by-date=false), WSDL supports all HTTP verbs and it is now considered to be an acceptable method of documenting REST services.

The second alternative is **WADL**, the Web Application Description Language. WADL is championed by Sun Microsystems. Like the rest of REST, WADL is lightweight, easier to understand and easier to write than WSDL. In some respects, it is not as flexible as WSDL (no binding to SMTP servers), but it is sufficient for any REST service and much less verbose.

Here is a fragment from a WADL specification, describing Amazon's "Item Search" service:

<method name="GET" id="ItemSearch">

<request>

<param name="Service" style="query"

fixed="AWSECommerceService"/>

<param name="Version" style="query" fixed="2005-07-26"/>

<param name="Operation" style="query" fixed="ItemSearch"/>

<param name="SubscriptionId" style="query"

type="xsd:string" required="true"/>

<param name="SearchIndex" style="query"

type="aws:SearchIndexType" required="true">

<option value="Books"/>

<option value="DVD"/>

<option value="Music"/>

</param>

<param name="Keywords" style="query"

type="aws:KeywordList" required="true"/>

<param name="ResponseGroup" style="query"

type="aws:ResponseGroupType" repeating="true">

<option value="Small"/>

<option value="Medium"/>

<option value="Large"/>

<option value="Images"/>

</param>

</request>

<response>

<representation mediaType="text/xml"

element="aws:ItemSearchResponse"/>

</response>

</method>

As you can see, for format is mostly self-explanatory, and it enriches REST with such goodies as type safety using XML schema types.

* The entire document is only about 10 lines longer than this fragment (including XML namespace specifications, importing schema grammers, etc.) and can be found in [the WADL specification](https://wadl.dev.java.net/wadl20061109.pdf).
* For another real-world WADL document, check out [the W3C Unicorn project's](http://www.w3.org/QA/Tools/Unicorn/)specification of the [CSS validator service](http://www.w3.org/QA/Tools/Unicorn/contract/css-validator.wadl).

Some REST advocates, however, find even the lightweight WADL to be an overkill. And indeed, most REST services are documented by no more than a textual description (a human-readable HTML file).

BY DR. M. ELKSTEIN [3 COMMENTS](http://www.blogger.com/comment.g?blogID=6432484966567158349&postID=5615184770241228601)

[12. REST Examples in Different Languages](http://rest.elkstein.org/2008/02/rest-examples-in-different-languages.html)

The following sections provide brief examples for accessing REST services in different programming languages. More than anything else, they present mechanisms for sending HTTP GET and/or POST requests in each language; the rest of using REST is simply a matter of architectural design.

You might wish to skip directly to your language of choice (the list is alphabetical):

* [C#](http://rest.elkstein.org/2008/02/using-rest-in-c-sharp.html)
* [Java](http://rest.elkstein.org/2008/02/using-rest-in-java.html)
* [JavaScript](http://rest.elkstein.org/2008/02/using-rest-in-javascript.html)
* [Perl](http://rest.elkstein.org/2008/02/using-rest-in-perl.html)
* [PHP](http://rest.elkstein.org/2008/02/using-rest-in-php.html)
* [Python](http://rest.elkstein.org/2008/02/using-rest-in-python.html)
* [Ruby](http://rest.elkstein.org/2008/02/using-rest-in-ruby.html)
* [Groovy](http://rest.elkstein.org/2008/02/using-rest-in-groovy.html)

Finally, [the tutorial conclusion](http://rest.elkstein.org/2008/02/for-more-about-rest.html) contains references to additional resources.

[Using REST in Python](http://rest.elkstein.org/2008/02/using-rest-in-python.html)

Issuing GET Requests  
The Python module [urllib2](http://docs.python.org/lib/module-urllib2.html) makes reading URLs trivial:

import urllib2

url = 'http://www.acme.com/products/3322'

response = urllib2.urlopen(url).read()

Errors are reported as exceptions (urllib2.HTTPError or urllib2.URLError).

Issuing POST Requests  
A POST request is just as easy, simply passing (encoded) request data as an extra parameter to urlopen, thus:

import urllib

import urllib2

url = 'http://www.acme.com/users/details'

params = urllib.urlencode({

'firstName': 'John',

'lastName': 'Doe'

})

response = urllib2.urlopen(url, params).read()

Note that encoding is done using a function from the urllib module.