**REST vs SOAP, the difference between soap and rest**



Someone asked me a question today “Why would anyone choose SOAP ([Simple Object Access Protocol](http://en.wikipedia.org/wiki/SOAP)) instead of REST ([Representational State Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer))?” My response: “The general rule of thumb I’ve always heard is ‘Unless you have a definitive reason to use SOAP use REST’”. He asked “what’s one reason?” I thought about it for a minute and honestly answered that I haven’t ever come across a reason. My background is building great internet companies.

While he seemed satisfied, I wasn’t very happy with that answer, I did some homework and here’s my summary on REST versus SOAP, the difference between SOAP and REST and why anyone would choose SOAP. As usual, with competing technologies both have value, the challenge is to know when to use each one (spoiler: luckily the answer is almost always REST).

I’m clearly boiling down a somewhat so please don’t flame me for simplifying things, but feel free to provide any corrections you feel are necessary.

**Definitions**

**REST**

RESTs sweet spot is when you are exposing a public API over the internet to handle CRUD operations on data. REST is focused on accessing named resources through a single consistent interface.

**SOAP**

SOAP brings it’s own protocol and focuses on exposing pieces of application logic (not data) as services. SOAP exposes operations. SOAP is focused on accessing named operations, each implement some business logic through different interfaces.

Though SOAP is commonly referred to as “[web services](http://en.wikipedia.org/wiki/Web_service)” this is a misnomer. SOAP has very little if anything to do with the Web. REST provides true “Web services” based on URIs and HTTP.

By way of illustration here are few calls and their appropriate home with commentary.

getUser(User);

This is a rest operation as you are accessing a resource (data).

switchCategory(User, OldCategory, NewCategory)

This is a SOAP operation as you are performing an operation.

Yes, either could be done in either SOAP or REST. The purpose is to illustrate the conceptual difference.

**Why REST?**

Here are a few reasons why REST is almost always the right answer.

Since REST uses standard HTTP it is much simpler in just about ever way. Creating clients, developing APIs, the documentation is much easier to understand and there aren’t very many things that REST doesn’t do easier/better than SOAP.

REST permits many different data formats where as SOAP only permits XML. While this may seem like it adds complexity to REST because you need to handle multiple formats, in my experience it has actually been quite beneficial. JSON usually is a better fit for data and parses much faster. REST allows better support for browser clients due to it’s support for JSON.

REST has better performance and scalability. REST reads can be cached, SOAP based reads cannot be cached.

It’s a bad argument (by authority), but it’s worth mentioning that Yahoo uses REST for all their services including Flickr and del.ici.ous. Amazon and Ebay provide both though Amazon’s internal usage is nearly all REST [source](http://www.oreillynet.com/pub/wlg/3005). Google used to provide only SOAP for all their services, but in 2006 they deprecated in favor of REST [source](http://code.google.com/apis/soapsearch/). It’s interesting how there has been an internal battle between rest vs soap at amazon. For the most part REST dominates their architecture for web services.

**Why SOAP?**

Here are a few reasons you may want to use SOAP.

**WS-Security**

While SOAP supports SSL (just like REST) it also supports WS-Security which adds some enterprise security features. Supports identity through intermediaries, not just point to point (SSL). It also provides a standard implementation of data integrity and data privacy. Calling it “Enterprise” isn’t to say it’s more secure, it simply supports some security tools that typical internet services have no need for, in fact they are really only needed in a few “enterprise” scenarios.

**WS-AtomicTransaction**

Need ACID Transactions over a service, you’re going to need SOAP. While REST supports transactions, it isn’t as comprehensive and isn’t ACID compliant. Fortunately ACID transactions almost never make sense over the internet. REST is limited by HTTP itself which can’t provide two-phase commit across distributed transactional resources, but SOAP can. Internet apps generally don’t need this level of transactional reliability, enterprise apps sometimes do.

**WS-ReliableMessaging**

Rest doesn’t have a standard messaging system and expects clients to deal with communication failures by retrying. SOAP has successful/retry logic built in and provides end-to-end reliability even through SOAP intermediaries.

**Summary**

In Summary, SOAP is clearly useful, and important. For instance, if I was writing an iPhone application to interface with my bank I would definitely need to use SOAP. All three features above are required for banking transactions. For example, if I was transferring money from one account to the other, I would need to be certain that it completed. Retrying it could be catastrophic if it succeed the first time, but the response failed.

[Why SOAP sucks](http://www.somebits.com/weblog/tech/bad/whySoapSucks.html)

There's an [amusing dialogue](http://wanderingbarque.com/nonintersecting/2006/11/15/the-s-stands-for-simple/) floating around about how simple SOAP is. As someone who bears some past responsibility for well used SOAP services (Google's APIs for [search](http://code.google.com/apis/soapsearch/index.html) and [AdWords](http://www.google.com/apis/adwords/)) let me say now I'd never choose to use SOAP and WSDL again. I was wrong.

The promise of SOAP and WSDL was removing all the plumbing. When you look at SOAP client examples, they're two lines of code. "Generate proxy. RPC to proxy." And for toys, that actually works. But for serious things it doesn't. I don't have the space to explain all the problems right now (if you've seen my talks at O'Reilly conferences, you know), but they boil down to massive interoperability problems. Good lord, you can't even pass a *number* between languages reliably, much less arrays, or dates, or structures that can be null, or... It just doesn't work. Maybe with enough effort SOAP interop could eventually be made to work. It's not such a problem if you're writing both the client and the server. But if you're publishing a server for others to use? Forget it.

The deeper problem with SOAP is strong typing. WSDL accomplishes its magic via XML Schema and strongly typed messages. But strong typing is a bad choice for loosely coupled distributed systems. The moment you need to change anything, the type signature changes and all the clients that were built to your earlier protocol spec break. And I don't just mean major semantic changes break things, but cosmetic things like accepting a 64 bit int where you use used to only accept 32 bit ints, or making a parameter optional. SOAP, in practice, is incredibly brittle. If you're building a web service for the world to use, you need to make it flexible and loose and a bit sloppy. Strong typing is the wrong choice.

The REST / HTTP+POX services typically assume that the clients will be flexible and can make sense of messages, even if they change a bit. And in practice this seems to work pretty well. My favourite API to use is the [Flickr API](http://www.flickr.com/services/api/), and my favourite client for it is [48 lines of code](http://www.somebits.com/weblog/tech/good/flickrClient.html). It supports 100+ Flickr API methods. How? Fast and loose. And it works great.

To be fair, SOAP can be forced to work. Using SOAP didn't really hurt adoption of the APIs I worked on. But it sure didn't help either. So what's the alternative? I'm not sure. Right now I'd probably go the HTTP+POX route while trying to name my resources well enough that the REST guys will invite me to their parties. But XML itself is [such](http://www.somebits.com/weblog/tech/xmlDumb.html) a [disaster](http://www.somebits.com/weblog/tech/bad/xmlCode.html) and AJAX is starting to show the cracks in HTTP (like, say, the lack of asynchrony).

Truly, none of this protocol fiddling matters. Just do something that works.

Simple Object Access Protocol (SOAP) and REpresentational State Transfer (REST) are two answers to the same question: how to access Web services. The choice initially may seem easy, but at times it can be surprisingly difficult.

SOAP is a standards-based Web services access protocol that has been around for a while and enjoys all of the benefits of long-term use. Originally developed by Microsoft, SOAP really isn’t as simple as the acronym would suggest.

REST is the newcomer to the block. It seeks to fix the problems with SOAP and provide a truly simple method of accessing Web services. However, sometimes SOAP is actually easier to use; sometimes REST has problems of its own. Both techniques have issues to consider when deciding which protocol to use.

Before I go any further, it’s important to clarify that while both SOAP and REST share similarities over the HTTP protocol, SOAP is a more rigid set of messaging patterns than REST. The rules in SOAP are important because without these rules, you can’t achieve any level of standardization. REST as an architecture style does not require processing and is naturally more flexible. Both SOAP and REST rely on well-established rules that everyone has agreed to abide by in the interest of exchanging information.

## A Quick Overview of SOAP

SOAP relies exclusively on XML to provide messaging services. Microsoft originally developed SOAP to take the place of older technologies that don’t work well on the Internet such as the Distributed Component Object Model (DCOM) and Common Object Request Broker Architecture (CORBA). These technologies fail because they rely on binary messaging; the XML messaging that SOAP employs works better over the Internet.

After an initial release, Microsoft submitted SOAP to the Internet Engineering Task Force (IETF) where it was standardized. SOAP is designed to support expansion, so it has all sorts of other acronyms and abbreviations associated with it, such as WS-Addressing, WS-Policy, WS-Security, WS-Federation, WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction, and WS-RemotePortlets. In fact, you can find a whole laundry list of these standards on [Web Services Standards](http://www1.innoq.com/soa/ws-standards/poster/).

The point is that SOAP is highly extensible, but you only use the pieces you need for a particular task. For example, when using a public Web service that’s freely available to everyone, you really don’t have much need for WS-Security.

The XML used to make requests and receive responses in SOAP can become extremely complex. In some programming languages, you need to build those requests manually, which becomes problematic because SOAP is intolerant of errors. However, other languages can use shortcuts that SOAP provides; that can help you reduce the effort required to create the request and to parse the response. In fact, when working with .NET languages, you never even see the XML.

Part of the magic is the Web Services Description Language (WSDL). This is another file that’s associated with SOAP. It provides a definition of how the Web service works, so that when you create a reference to it, the IDE can completely automate the process. So, the difficulty of using SOAP depends to a large degree on the language you use.

One of the most important SOAP features is built-in error handling. If there’s a problem with your request, the response contains error information that you can use to fix the problem. Given that you might not own the Web service, this particular feature is extremely important; otherwise you would be left guessing as to why things didn’t work. The error reporting even provides standardized codes so that it’s possible to automate some error handling tasks in your code.

An interesting SOAP feature is that you don’t necessarily have to use it with the HyperText Transfer Protocol (HTTP) transport. There’s an actual specification for [using SOAP over Simple Mail Transfer Protocol](http://www.pocketsoap.com/specs/smtpbinding/) (SMTP) and there isn’t any reason you can’t use it over other transports. In fact, developers in some languages, such as Python, are [doing just that](http://www.ibm.com/developerworks/webservices/library/ws-pyth12/index.html).

## A Quick Overview of REST

Many developers found SOAP cumbersome and hard to use. For example, working with SOAP in JavaScript means writing a ton of code to perform extremely simple tasks because you must create the required XML structure absolutely every time.

REST provides a lighter weight alternative. Instead of using XML to make a request, REST relies on a simple URL in many cases. In some situations you must provide additional information in special ways, but most Web services using REST rely exclusively on obtaining the needed information using the URL approach. REST can use four different HTTP 1.1 verbs (GET, POST, PUT, and DELETE) to perform tasks.

Unlike SOAP, REST doesn’t have to use XML to provide the response. You can find REST-based Web services that output the data in Command Separated Value (CSV), JavaScript Object Notation (JSON) and Really Simple Syndication (RSS). The point is that you can obtain the output you need in a form that’s easy to parse within the language you need for your application.

As an example of working with REST, you could create a URL for [Weather Underground](http://www.wunderground.com/weather/api). The [API’s documentation page](http://www.wunderground.com/weather/api/d/docs) shows an example URL of http://api.wunderground.com/api/Your\_Key/conditions/q/CA/San\_Francisco.json. The information you receive in return is a JSON formatted document containing the weather for San Francisco. You can use your browser to interact with the Web service, which makes it a lot easier to create the right URL and verify the output you need to parse with your application.

## Deciding Between SOAP and REST

Before you spend hours fretting over the choice between SOAP and REST, consider that some Web services support one and some the other. Unless you plan to create your own Web service, the decision of which protocol to use may already be made for you. Extremely few Web services, such as Amazon, support both. The focus of your decision often centers on which Web service best meets your needs, rather than which protocol to use.

### Soap Vs Rest

SOAP is definitely the heavyweight choice for Web service access. It provides the following advantages when compared to REST:

* Language, platform, and transport independent (REST requires use of HTTP)
* Works well in distributed enterprise environments (REST assumes direct point-to-point communication)
* Standardized
* Provides significant pre-build extensibility in the form of the WS\* standards
* Built-in error handling
* Automation when used with certain language products

REST is easier to use for the most part and is more flexible. It has the following advantages when compared to SOAP:

* No expensive tools require to interact with the Web service
* Smaller learning curve
* Efficient (SOAP uses XML for all messages, REST can use smaller message formats)
* Fast (no extensive processing required)
* Closer to other Web technologies in design philosophy

### Locating Free Web Services

The best way to discover whether SOAP or REST works best for you is to try a number of free Web services. Rolling your own Web service can be a painful process, so it’s much better to make use of someone else’s hard work. In addition, as you work with these free Web services you may discover that they fulfill a need in your organization, and you can save your organization both time and money by using them. Here are some to check out:

* [Free-Web-Services.com](http://free-web-services.com/)
* [WebserviceX.NET](http://www.webservicex.net/ws/wscatlist.aspx)
* [Yahoo! Directory of Web Services](http://dir.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/Web_Services/)
* [XMethods](http://www.xmethods.net/ve2/index.po)

One common concern about using a free Web service is the perception that it could somehow damage your system or network. Web services typically send you text, not scripts, code, or binary data, so the risks are actually quite small.

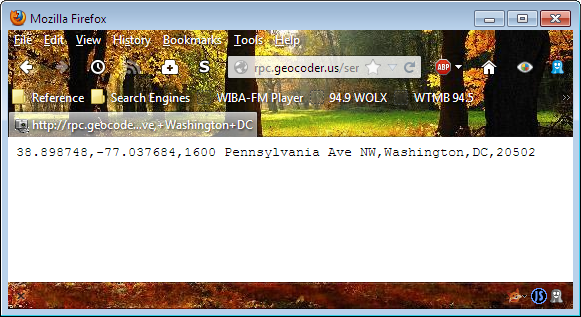
Of course, there’s also the concern that Web services will disappear overnight. In most cases, these Web services are exceptionally stable and it’s unlikely that any of them will disappear anytime soon. I’ve been using some of them now for five years without any problem. However, stick with Web services from organizations with a large Internet presence. Research the Web service before you begin using it.

### Working with the Geocoder Web Service

To make it easier to understand how SOAP and REST compare, I decided to provide examples of both using the same free Web service, [geocoder.us](http://geocoder.us/help/) (thank you to [Mark Yuabov](https://plus.google.com/114952462845494608775/posts) for suggesting it). This simple Web service accepts an address as input and spits out a longitude and latitude as output. You could probably mix it with the Google Maps API example I present in “[Using the Google Maps API to Add Cool Stuff to Your Applications](http://blog.smartbear.com/how-to/using-the-google-maps-api-to-add-cool-stuff-to-your-applications/).”

## Viewing a Simple REST Example

Sometimes, simple is best. In this case, REST is about as simple as it gets because all you need is an URL. Open your browser—it doesn’t matter which one—and type http://rpc.geocoder.us/service/csv?address=1600+Pennsylvania+Ave,+Washington+DC in the address field. Press Enter. You’ll see the output in your browser in CSV format:

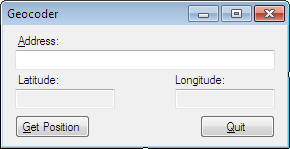


You see the latitude, followed by the longitude, followed by the address you provided. This simple test works for most addresses in most major cities (it doesn’t work too well for rural addresses, but hey, what do you expect for free?). The idea is that you obtain the latitude and longitude needed for use with other Web services. By combining Web services together with a little glue code, you can create really interesting applications that do amazing things in an incredibly short time with little effort on your part. Everyone else is doing the heavy lifting.

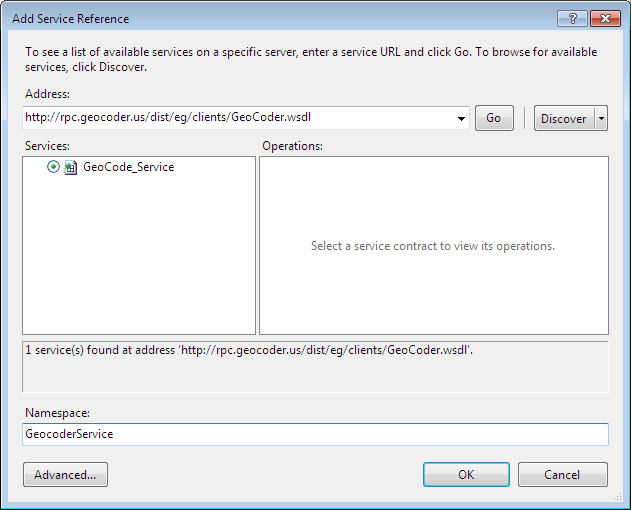
## Explaining a Simple SOAP Example

SOAP, by its very nature, requires a little more setup, but I think you’ll be amazed at how simple it is to use.

Begin this example by creating Windows Forms application using [Visual Studio](http://smartbear.com/products/qa-tools/application-performance-profiling/profiling-in-microsoft-visual-studio). The sample code uses C#, but the same technique works fine with other .NET languages (you’ll need to modify the code to fit). Add labels, textboxes, and buttons as shown here (the Latitude and Longitude fields are read-only).



Here’s where the automation comes into play. Right click References in Solution Explorer and choose Add Service Reference from the context menu. You’ll see the Add Service Reference dialog box. Type the following address into the address field: http://rpc.geocoder.us/dist/eg/clients/GeoCoder.wsdl and click Go. Type GeocoderService in the namespace field. Your dialog box should look like the one shown here.



Click OK. Visual Studio adds the code needed to work with Geocoder in the background.

At this point, you’re ready to use the Web service. All you need to do is to add some code to the Get Position button as shown here.

private void btnGetPosition\_Click(object sender, EventArgs e)  
{  
   // Create the client.  
   GeocoderService.GeoCode\_PortTypeClient Client =  
      new GeocoderService.GeoCode\_PortTypeClient();

   // Make the call.  
   GeocoderService.GeocoderResult[] Result =  
      Client.geocode(txtAddress.Text);

   // Check for an error result.  
   if (Result != null)  
   {  
      // Display the results on screen.  
      txtLatitude.Text = Result[0].lat.ToString();  
      txtLongitude.Text = Result[0].@long.ToString();  
   }  
   else  
   {  
      // Display an error result.  
      txtLatitude.Text = "Error";  
      txtLongitude.Text = "Error";  
   }  
}

The code begins by creating a client. This is a common step for any Web service you use with Visual Studio (or other environments that support SOAP natively). To see another version of the same step, check out [the PHP example](http://geocoder.us/help/php_kenny.shtml).

After you create the client, you use it to call one of the methods supported by the Web service. In this case, you call geocode() and pass the address you want to work with. The result of the call is stored in a GeocoderResult variable named Result. A single address could possibly end up providing multiple positions if you aren’t specific enough, so this information is passed back as an array.

Let’s assume that no errors occur (resulting in a null return value). The example assumes that you provided great information, so it places the information found in the first Result entry into the Latitude and Longitude output. So, this example isn’t really that complicated compared with REST, but as you can see, even a simple example is more work.

## The Bottom Line: When To Use SOAP Or REST

Some people try to say that one process is better than the other, but this statement is incorrect. Each protocol has definite advantages and equally problematic disadvantages. You need to select between SOAP and REST based on the programming language you use, the environment in which you use it, and the requirements of the application. Sometimes SOAP is a better choice and other times REST is a better choice. In order to avoid problems later, you really do need to chart the advantages and disadvantages of a particular solution in your specific situation.

There’s one absolute you should get from this article. Don’t reinvent the wheel. It’s amazing to see companies spend big bucks to create Web services that already exist (and do a better job than the Web service the company creates). Look for free alternatives whenever possible. In many cases, the choice of Web service also determines your choice of protocol.

**See also:**

* [Riding The New Wave Of API’s](http://blog.smartbear.com/iot-2/riding-the-new-wave-with-api-monitoring-video/)
* [Are We in the Golden Age of APIs?](http://blog.smartbear.com/apis/are-we-in-the-golden-age-of-apis/)
* [Erik’s Quick Tip: Using MockService in soapUI](http://blog.smartbear.com/soapui/eriks-quick-tip-using-mockservice-in-soapui/)
* [Will the Next Web Platform Please Hold Still?](http://blog.smartbear.com/programming/will-the-next-web-platform-please-hold-still/)
* [Understanding The Open Web Stack](http://blog.smartbear.com/architecture/understanding-the-open-web-stack/)

SOAP and REST can't be compared directly, since the first is a protocol (or at least tries to be) and the second is an architectural style. This is probably one of the sources of confusion around it, since people tend to call REST any HTTP API that isn't SOAP.

Pushing things a little and trying to establish a comparison, the main difference between SOAP and REST is the degree of coupling between client and server implementations. A SOAP client works like a custom desktop application, tightly coupled to the server. There's a rigid contract between client and server, and everything is expected to break if either side changes anything. You need constant updates following any change.

A REST client is more like a browser. It's a generic client that knows how to use a protocol and standardized methods, and an application has to fit inside that. You don't violate the protocol standards by creating extra methods, you leverage on the standard methods and create the actions with them on your media type. If done right, there's less coupling, and changes can be dealt with more gracefully. A client is supposed to enter a REST service with zero knowledge of the API, except for the entry point and the media type. In SOAP, the client needs previous knowledge on everything he will be using, or it won't even begin the interaction. Additionally, a REST client can be extended by code-on-demand supplied by the server itself, the classical example being javascript code used to drive the interaction with another service on the client-side.

I think these are the crucial points to understand what REST is about, and how it differs from SOAP:

* REST is protocol independent. It's not coupled to HTTP. Pretty much like you can follow an ftp link on a website, a REST application can use any protocol for which there is an standardized URI scheme.
* REST is not mapping CRUD to HTTP methods. Read [this](http://stackoverflow.com/questions/19843480/s3-rest-api-and-post-method/19844272#19844272) answer for a detailed explanation on that.
* REST is as standardized as the parts you're using. Security and authentication in HTTP is standardized, so that's what you use when doing REST over HTTP.
* REST is not REST without [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS). This means a client only knows the entry point URI and the resources are supposed to return links the client should follow. Those fancy documentation generators that give URI patterns for everything you can do in a REST API miss the point completely. They are not only documenting something that's supposed to be following the standard, but when you do that, you're coupling the client to one particular moment in the evolution of the API, and any changes on the API have to be documented and applied, or it will break.
* REST is the architectural style of the web itself. When you enter Stack Overflow, you know what an User, a Question and an Answer are, you know the media types, and the website provides you with the links to them. A REST API has to do the same. If we designed the web the way people think REST should be done, instead of having a home page with links to Questions and Answers, we'd have a static documentation explaining that in order to view a question, you have to take the URI stackoverflow.com/questions/<id>, replace id with the Question.id and paste that on your browser. That's nonsense, but that's what many people think REST is.

This last point can't be emphasized enough. If your clients are building URIs from templates in documentation and not getting links in the resource representations, that's not REST. Roy Fielding, the author of REST, made it clear on this blog post: [REST APIs must be hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

With the above in mind, you'll realize that while REST might not be restricted to XML, to do it correctly with any other format you'll have to design and standardize some format for your links. Hyperlinks are standard in XML, but not in JSON. There are draft standards for JSON, like [HAL](http://stateless.co/hal_specification.html).

Finally, REST isn't for everyone, and a proof of that is how most people solve their problems very well with the HTTP APIs they call REST and never venture beyond that. REST is hard to do sometimes, especially in the beginning, but it pays over time with easier evolution on the server side, and client's resilience to changes. If you need something done quickly and easily, don't bother about getting REST right. It's probably not what you're looking for. If you need something that will have to stay online for years or even decades, then REST is for you.

EST vs SOAP is ***not*** the right question to ask.

REST, unlike SOAP is ***not*** a protocol.

REST is an architectural style and a design for network-based software architectures.

REST concepts are referred to as resources. A representation of a resource must be stateless. It is represented via some media type. Some examples of media types include XML, JSON, and RDF. Resources are manipulated by components. Components request and manipulate resources via a standard uniform interface. In the case of HTTP, this interface consists of standard HTTP ops e.g. GET, PUT, POST, DELETE.

@Abdulaziz's question does illuminate the fact that REST and HTTP are often used in tandem. This is primarily due to the simplicity of HTTP and its very natural mapping to RESTful principles.

### Fundamental REST Principles

**Client-Server Communication**

Client-server architectures have a very distinct separation of concerns. All applications built in the RESTful style must also be client-server in princple.

**Stateless**

Each client request to the server requires that its state be fully represented. The server must be able to completely understand the client request without using any server context or server session state. It follows that all state must be kept on the client. We will discuss stateless representation in more detail later.

**Cacheable**

Cache constraints may be used, thus enabling response data to to be marked as cacheable or not-cachable. Any data marked as cacheable may be reused as the response to the same subsequent request.

**Uniform Interface**

All components must interact through a single uniform interface. Because all component interaction occurs via this interface, interaction with different services is very simple. The interface is the same! This also means that implementation changes can be made in isolation. Such changes, will not affect fundamental component interaction because the uniform interface is always unchanged. One disadvantage is that you are stuck with the interface. If an optimization could be provided to a specific service by changing the interface, you are out of luck as REST prohibits this. On the bright side, however, REST is optimized for the web, hence incredible popularity of REST over HTTP!

The above concepts represent defining characteristics of REST and differentiate the REST architecture from other architectures like web services. It is useful to note that a REST service is a web service, but a web service is not necessarily a REST service.

See this blog [post](http://www.carminedimascio.com/2013/09/06/restful-design-principles/) on **REST Design Principals** for more details on **REST** and the above stated bullets.

|  |  |
| --- | --- |
| 7 down vote | SOAP (**Simple Object Access Protocol**) and REST (**Representation State Transfer**) both are beautiful in their own way. So I am not comparing them, instead, I am trying to depict the picture, when I preferred to use REST and when SOAP.  **What is payload?**  When data is sent over the Internet, each unit transmitted includes both header information and the actual data being sent. The header identifies the source and destination of the packet, **while the actual data is referred to as the payload**. In general, payload is the data that is carried on behalf of an application and the data received by the destination system.  Now for example I have to send a **Telegram** and we all know that the cost of the telegram will depend on number of words.  *So tell me among below mentioned these two messages, which one is cheaper to send?*  <name>Arin</name> |

or

"name": "Arin"

I know your answer will be second one although both representing the same message second one is cheaper in terms of cost.

So I am trying to say that, **sending data over the network in Json format is cheaper than sending it in Xml format in terms of payload**.

**Here is the first benefit or advantages of REST over SOAP**. SOAP only support XML, but REST supports different format like text, JSON, XML etc. And we already know, if we use Json then definitely we will be in better place in terms of payload.

Now, SOAP supports only XML, **but it also has its own advantages.**

**Really! How?**

SOAP relies on XML in three ways Envelope – that defines what is in the message and how to process it.

A set of encoding rules for data types, and finally the layout of the procedure calls and responses gathered.

This envelope is sent via a transport (HTTP/HTTPS), and an RPC (Remote Procedure Call) is executed and the envelope is returned with information in a XML formatted document.

Here important point is that **one of the advantages of SOAP** is the use of the **“generic” transport** but **REST uses HTTP/HTTPS**. SOAP can use almost any transport to send the request but REST cannot. So here we got an advantage of using SOAP.

As I already mentioned in above paragraph **“REST uses HTTP/HTTPS”**, so go bit deeper on these words.

When we are talking about REST over HTTP, all security measures applied HTTP are inherited and this is known as **transport level security** and it secures messages only while **it is inside the wire** but once you delivered it on the other side you don’t really know how many stages it will have to go through before reaching the real point where the data will be processed. And of course all those stages could use something different than HTTP.**So Rest is not safer completely, right?**

But SOAP **supports SSL** just like REST additionally **it also supports WS-Security** which adds some enterprise security features. WS-Security offers protection from the **creation of the message to it’s consumption**. So for transport level security whatever loophole we found that can be prevented using WS-Security.

Apart from that, as **REST is limited by it's HTTP protocol** so it’s transaction support is neither ACID compliant nor can provide two phase commit across distributed transnational resources.

But SOAP has comprehensive support for both **ACID based transaction management** for short-lived transactions and compensation based transaction management for long-running transactions. It also supports **two-phase commit across distributed resources**.

I am not drawing any conclusion, but I will definitely prefer SOAP based web service while security, transaction etc are the main concerns.

Here is the "The Java EE 6 Tutorial" where they have said [A RESTful design may be appropriate when the following conditions are met](https://docs.oracle.com/javaee/6/tutorial/doc/giqsx.html). Have a look.

# RESTful Design Principles

by [cdimascio](http://carminedimascio.com/author/cdimascio/) · September 6, 2013

Here, we will outline the set of RESTful design principles that should be adhered to when creating a ‘proper’ RESTful service.

Let’s start with the basics. **What is REST**?

**REST** = REpresentational State Transfer. REST is an architectural style for network based software that requires stateless, cacheable, client-server communication via a uniform interface between components.

The primary focus of this blog post is to introduce REST along with REST terminology, REST concepts, and some simple examples describing what REST looks like in practice. As a secondary focus, I will address a topic that often confuses folks. Many folks often ask to compare REST vs SOAP. This comparison does not make sense. REST is an architectural style, while SOAP, like HTTP, is communication protocol. What does this mean? Well, REST and SOAP are not mutually exclusive. In theory, they can be used together. I would highly recommend against this usage, but their is nothing about the REST style that prohibits this case. During this post we’ll touch on this. We’ll also see why REST is so widely used over HTTP. All in all, the primary focus will remain an introduction to REST!

Let’s get started. First off, I introduced a number of loaded terms in my one line description of REST. These terms were stateless, cacheable, client-server communication, and uniform interface. These represent the basic principles of REST. Let’s briefly introduce these principles and their meaning within the context of REST.

**Basic Principles**

**Client-Server Communication**

Client-server architectures have a very distinct separation of concerns. All applications built in the RESTful style must also be client-server in princple.

**Stateless**

Each each client request to the server requires that its state be fully represented. The server must be able to completely understand the client request without using any server context or server session state. It follows that all state must be kept on the client. We will discuss stateless representation in more detail later.

**Cacheable**

Cache constraints may be used, thus enabling response data to to be marked as cacheable or not-cachable. Any data marked as cacheable may be reused as the response to the same subsequent request.

**Uniform Interface**

All components must interact through a single uniform interface. Because all component interaction occurs via this interface, interaction with different services is very simple. The interface is the same! This also means that implementation changes can be made in isolation. Such changes, will not affect fundamental component interaction because the uniform interface is always unchanged. One disadvantage is that you are stuck with the interface. If an optimization could be provided to a specific service by changing the interface, you are out of luck as REST prohibits this. On the bright side, however, REST is optimized for the web, hence incredible popularity of REST over HTTP!

The above concepts represent defining characteristics of REST and differentiate the REST architecture from other architectures like web services. It is useful to note that a REST service is a web service, but a web service is not necessarily a REST service.

Let’s now dive into a bit more detail and discuss a variety of elements used to compose a RESTful system.

**Resource and Resource Identifier:**

A key abstraction of REST is the resource. A resource can be just about anything. It can be a document or an image, an object, a collection of other resources, and more.  A resource is identify by its resource identifier. The resource identifier is often used when multiple components communicate with one another. They are able to reference specific resources using the resource identifier.

In practice, resources are nouns. Resources and identified by URIs  
e.g. This Car resource is identified by the resource identifier, http://www.automart.com/cars/12345

**Representation:**

Components perform actions a resource by applying operation provided by the component’s uniform interface. A resource is represented by its current state or its intended state (assuming the action will modify the resource in some way). This representation includes a sequence of bytes and some description of those bytes. The format of a representation is defined as its media type.

In practice, resources are often represented as XML, JSON, RDF, and more

**Basic Principles in Practice**

Let’s harken back to the Basic Principles section and describe how those principles can be applied in practice.

**Client-server**

HTTP is a client-server protocol. Why not use it with REST. Check!

**Uniform Interface**

REST is optimized for the web, thus HTTP is typically used. HTTP defines GET,POST, PUT, DELETE. Woah! That meets REST’s requirement to provide a uniform interface for components.

**Cacheable**

HTTP provides a cache control mechanism. See [here](http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html). Dang! HTTP just filled another REST requirement

**Stateless**

Hmm. Not quite so easy. Let’s apply some rules to the uniform interface provided by HTTP

GET – Safe, Cacheable, Idempotent  
PUT – Idempotent  
DELETE – Idempotent  
HEAD – Safe, Idempotent  
POST – n/a

Cool! But what does that mean?

– Safe – the operation must not have side effects  
– Cacheable – the result may be cached e.g. by a proxy server  
– Idempotent – The operation must always return the same result

Check!

**Rest Practical Usage**

Let’s now provide some example of in-practice usage:

**Resource and Resource Identifiers**

Example of resources are Car, Engine, Part. Each resource is identified by its *resource identifier*. For example:

Car: http://www.automart.com/cars/12345

Part: http://www.automart.com/part/12345

Part: http://www.automart.com/engine/12345

**Representation**  
Our Car with *resource identifier,* http://www.automart.com/cars/12343 can be manipulated by a component via a uniform interface of GET, POST, PUT, DELETE. Here are some examples:

GET returns a representation of a resource’s state, For example, http://www.automart.com/cars/12343.  
An XML representation of that state might be:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <Car>    <Make>Audi</Make>    <Model>A5</Model>    <Year>2013</Year>  </Car> |

or in JSON

|  |  |
| --- | --- |
| 1  2  3  4  5 | {  "Make" : "Audi",  "Model" : "A5",  "Year" : 2013  } |

**Representation with Linked Resources**  
Resource representations may contain links to other resources  
e.g.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <Car>    ...    <Engine uri="<http://www.automart.com/engine/1242>"/>    ...  </Car> |

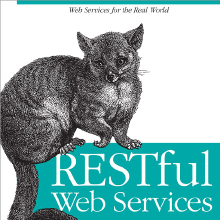
or in JSON

|  |  |
| --- | --- |
| 1  2  3  4  5 | {  ...  "engine" : "<http://www.automart.com/engine/1242>"  ...  } |

That’s it!!

# [REST vs HTTP+POX vs SOAP](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/)

November 6th, 2006 • [Related](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/) • [Filed Under](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/)

[](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/)

[Sam](http://www.intertwingly.net/blog/2006/11/05/POX-and-SOAP) and [Leonard](http://www.crummy.com/2006/11/05/2) ponder the differences among them. Here’s the deal:

1. REST == HTTP GET
2. HTTP+POX == HTTP GET & POST
3. SOAP == HTTP POST, with interop issues

When I’m talking to people about web services, and I hear them use various terms, this is what I feel they’re talking about most of the time.

**REST**

REST almost always indicates HTTP GET with a query string. The results are probably XML, but lately they could also be JSON.

No other HTTP verbs are used (especially PUT and DELETE), and it’s highly likely that all requests are routed through the same URI.

For example:

http://api.example.org/rest?action=search&query=trachtenberg

**HTTP+POX**

Nobody uses this term. However, there are some web services that, while not “true REST,” use more than HTTP GET. I place them in this category.

These likely exist because the service allows you to submit a large quantity of data that is not human readable, so it does not make sense to place as part of a query string. Blobs of HTML and pictures come to mind.

For example, the eBay “XML API”. You can submit items over eBay web services. Since the item description is pure HTML, it exceeded the maximum query string length our web server would accept at the time.

Thus, the eBay XML API was invented: you HTTP POST an XML document, we return an XML document.

Again, like REST, there’s no concept of resources. Everything goes through the same URI. The action is indicated either as part of the query string or in POST body. (Or both, as in the eBay case.)

**SOAP**

SOAP is identical to HTTP+POX, except that you’re required to use XML Namespaces and XML Schema.

You are also going to have interop issues when trying to generate the SOAP envelope to send or parse the SOAP envelope that’s returned.

XML Namespaces and XML Schema are minor headaches, but people are willing to deal with them. People understand that, in theory, namespaces are good, even if XML Namespaces are a little funky.

XML Schema can be confusing. However, it does allow you to validate the response, which most people find to be somewhat useful. The fact that XML Schema may not be the best way to describe XML data is a different issue.

If SOAP merely had these two issues, people would work through it. The real problem with SOAP is that the specification is so confusing, people can’t build interoperable clients and servers. This drives people mad.

People know how to generate arbitrary XML (it’s just text after all) and send a HTTP POST request. They also know how to parse XML, particularly when they know ahead of time what they should expect. (That’s why PHP 5’s SimpleXML extension is a great web services client.)

What they cannot do is decipher a WSDL file to determine what crazy combination of XML the server is expecting (especially when the server is not following the standard). And, assuming they can get through that, they cannot figure out how to repeatedly hammer their SOAP client into producing the magical combination of XML necessary to placate the SOAP server.

That’s the difference between HTTP+POX and SOAP.

It’s not the complexity. Well, it’s not complexity in terms of needing to build up a large document. It’s complexity in terms of trying to understand the hundreds of pages of XML specifications to generate the request and not having good examples of working XML documents to crib off of.

As to whether the message name should go in the URI? Nobody cares.

Popularity: 70% [[?](http://alexking.org/projects/wordpress/popularity-contest)]

### There Are 9 Responses So Far. [»](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/#respond)

1. http://www.gravatar.com/avatar.php?gravatar_id=e698f564ac90c4c248f1f678caafd624&size=48&rating=PG

Comment by [*Keith Gaughan*](http://talideon.com/) on [6 November 2006](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/#comment-19833):

Ok, that first one definitely isn’t REST: it’s HTTP+POX. Anyway, REST uses the gamut of HTTP, not just GET and isn’t just some HTTP based RPC mechanism (which the use of some parameter like “action”, “do” or so on would indicate). Whatever gave you that idea?

1. http://www.gravatar.com/avatar.php?gravatar_id=b83aaa5e5c8129987c8c54f2974e84d3&size=48&rating=PG

Comment by [*Leonard Richardson*](http://www.crummy.com/) on [6 November 2006](http://www.trachtenberg.com/blog/2006/11/06/rest-vs-httppox-vs-soap/#comment-19886):

I’m still digesting what you say about SOAP but I gotta disagree with those definitions of REST and HTTP+POX. (I don’t know whether you hold these opinions or are just reporting what others say when you discuss the matter.) They’re fairly common definitions and are remarkably close to what I used to think, so I think it’s worth taking them on.

Most of the web services you describe as REST are in fact REST, because a lot of public web services only let you fetch data. They use GET in a RESTful way because all their URIs are safe. But if a service lets you change the dataset it needs to use the other HTTP actions (PUT, DELETE, and POST, the latter of which still confuses me) for that. The best way to do this is to structure the URIs into resources, so that if you want to get the weblog entry at URI “/weblog/posts/100″, you send a GET request to that URI, and if you want to delete that entry you send a DELETE request to the same URI.

So the Amazon E-Commerce Service is almost entirely REST. It’s full of URI query-string options for searching and getting info from the Amazon product database,. Any query you can construct from the options describes a resource (a particular slice of the database) and GETting it gives you an XML depiction of that resource. But it also exposes some options called CartAdd, CartClear, etc. which store data in your shopping cart on the server side. This violates REST.

GET is for getting resources, and using it to make changes is dangerous. You may remember the flap that resulted when Google released a browser-side cache that followed links. It followed links like “/weblog/delete?id=100″ and triggered actions the user hadn’t requested. “Cart” can be considered a resource that you can add, modify, delete, etc. but “CartAdd” is an operation. And in fact you’ll see people saying that AECS is not REST for this reason.

Near as I can tell HTTP+POX was invented by REST eggheads to shift off a bunch of services that didn’t meet the REST criteria, like the del.icio.us API, the Flickr API, the non-RESTful part of AWS, and so on. But since the term was coined by REST fanatics as a wall around REST, there’s no clear line of demarcation on the other side.

HTTP+POX services and SOAP services both have rules about what XML documents you can send under what circumstances. You could have a pathological service that was more idiosyncratic than SOAP in the XML it accepted. I’d say that would be worse than SOAP, and I wouldn’t want to call it HTTP+POX because then SOAP would be HTTP+POX too. That’s why I’m looking for another way to make the distinction. There might be no sharp distinction at all, just a gradient that includes SOAP near the far end.

Send me and Sam email if you’d like to continue this discussion and help us with the definition