**What is a web service?**

W3C defintion for a web service is, “a software system designed to support interoperable machine-to-machine interaction over a network”.

Web service is an interface for your software. We may have web user interface or a thick client (desktop) user interface for our software. Imagine web service as another similar interface. This interface is not for humans but for softwares.

Web/thick client directly serves end user as an interface to interact with the software. Web service serves as an interface to software developers. Using web service as an API, developers can build external systems that will interact with the software.

There are two major categories of web services

1. SOAP Web Service
2. RESTful Web Service

**Soap Web Service**

Simple Object Access Protocol (SOAP) is a standard protocol specification for message exchange based on XML. Communication between the web service and client happens using XML messages. SOAP defines the rules for communication like what are all the tags that should be used in XML and their meaning.

**RESTful Web Service**

RESTful web service uses architectures that use HTTP or similar protocols by restricting the interface to use standard operations like GET, POST, PUT, DELETE for HTTP. Based on my experience RESTful is easier to develop. I know this statement will invite wrath of SOAP lovers.

**WSDL**

One major component of a web service is Web Services Description Language (WSDL). It is an xml file that describes the web service technically in a machine readable format. That is, using this WSDL file we can understand things like,

* Port / Endpoint – URL of the web service (using which we should access it)
* Input message format
* Output message format
* Security protocol that needs to be followed (like https)
* Which protocol the web service uses

### REST vs SOAP

With all the debate about SOAP vs REST I thought it would be worth a few column inches pointing out the differences between the two.

REST is an architectural style. SOAP is a message format. With SOAP there is a well defined processing and extensibility model. The two are not directly comparable in my opinion. The confusions comes as both approaches (for lack of a better word) are used to deliver web services.

In the future I would hope to see both REST and SOAP being offered (as they are in our very own Amazon Web Services AWS), there are peculiar advantages for each approach.

There is the simplicity, of REST with its HTTP verbs allowing for simple querying of a distant resource, and the coming transactional model support in SOAP, that will allow for far more complicated activities via Web Services.  
  
In fact using SOAP 1.2 many of the RESTful features people so love (e.g. URI interface) can be combined with a SOAP payload, I think this is the beginning of an beautiful friendship.

REST’s straight forward nature offers low barriers of entry into using web services, this has resulted in the typical take up being 80% skewed in favor of REST  implementations by our own AWS.

I tend to think of REST as an introduction to the world of web services, and introduction that’s mostly good enough, especially for now, but one where it may need to be combined with some of the heavy weight features being built into SOAP for future applications.

By now, most developers have at least, from a periphery, been exposed to the REST approach, which uses a standard URI (Uniform Resource Identifier) that makes a call to a web service like *http/https://www.mycompany.com/program/method?Parameters=xx*. The approach is very simple to understand and can be executed on really any client or server that has HTTP/HTTPS support. The command can execute using the HTTP Get method. So developers that use this approach, cite the ease of development, use of the existing web infrastructure, and little learning overhead as key advantages to the style.

However SOAP, the granddaddy of all web services interfaces, is not going away anytime soon, and in fact with the introduction of SOAP 1.2 has fixed many of the perceived short-comings of the technology and pushing it to new levels of both adoption and ease-of-use. It should also be noted that the acronym SOAP no longer stands for Simple Object Access Protocol as of the 1.2 specification from the W3C organization; it is now just the name of the specification.

Now keep in mind that using SOAP 1.2 has some additional overhead that is not found in the REST approach, but that overhead also has advantages. First, SOAP relies on XML (Extensible Markup Language) in three ways; the Envelope – that defines what is in the message and how to process it, a set of encoding rules for datatypes, 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.

It is important to note that one of the advantages of SOAP is the use of the “generic” transport. While REST today uses HTTP/HTTPS, SOAP can use almost any transport to send the request, using everything from the afore mentioned to SMTP (Simple Mail Transfer Protocol) and even JMS (Java Messaging Service). However, one perceived disadvantage is the use of XML because of the verboseness of it and the time it takes to parse.

However, the good news for web developers is that both technologies are very viable in today’s market. Both REST and SOAP can solve a huge number of web problems and challenges, and in many cases each can be made to do the developers bidding, which means they can work across the domain.

But the untold story is that both technologies can be mixed and matched. REST is very easy to understand and is extremely approachable, but does lack standards and is considered an architectural approach. In comparison, SOAP is an industry standard with a well-defined protocol and a set of well-established rules to be implemented, and it has been used in systems both big and small.

So this means areas that REST works really well for are:

* **Limited bandwidth and resources;** remember the return structure is really in any format (developer defined). Plus, any browser can be used because the REST approach uses the standard *GET*, *PUT*, *POST*, and *DELETE* verbs. Again, remember that REST can also use the *XMLHttpRequest* object that most modern browsers support today, which adds an extra bonus of AJAX.
* **Totally stateless operations;** if an operation needs to be continued, then REST is not the best approach and SOAP may fit it better. However, if you need stateless CRUD (Create, Read, Update, and Delete) operations, then REST is it.
* **Caching situations;** if the information can be cached because of the totally stateless operation of the REST approach, this is perfect.

That covers a lot of solutions in the above three. So why would I even consider SOAP? Again, SOAP is fairly mature and well-defined and does come with a complete specification. The REST approach is just that, an approach and is wide open for development, so if you have the following then SOAP is a great solution:

* **Asynchronous processing and invocation;** if your application needs a guaranteed level of reliability and security then SOAP 1.2 offers additional standards to ensure this type of operation. Things like WSRM – WS-Reliable Messaging.
* **Formal contracts;** if both sides (provider and consumer) have to agree on the exchange format then SOAP 1.2 gives the rigid specifications for this type of interaction.
* **Stateful operations**; if the application needs contextual information and conversational state management then SOAP 1.2 has the additional specification in the WS\* structure to support those things (Security, Transactions, Coordination, etc). Comparatively, the REST approach would make the developers build this custom plumbing.

[](http://www.ajaxonomy.com/)

## The Study of Ajax and Other Interesting Web Technologies

[Home](http://www.ajaxonomy.com/)

# Web Services, Part 1: SOAP vs. REST

By [Brennan Spies](http://www.ajaxonomy.com/users/Brennan%2BSpies" \o "View user profile.) - Posted on May 2nd, 2008

Tagged:

* [Web Services](http://www.ajaxonomy.com/tag/Web%2BServices" \o ")
* [XML](http://www.ajaxonomy.com/tag/XML" \o ")

Developers new to web services are often intimidated by parade of technologies and concepts required to understand it: REST, SOAP, WSDL, XML Schema, Relax NG, UDDI, MTOM, XOP, WS-I, WS-Security, WS-Addressing, WS-Policy, and a host of other WS-\* specifications that seem to multiply like rabbits. Add to that the Java specifications, such as JAX-WS, JAX-RPC, SAAJ, etc. and the conceptual weight begins to become heavy indeed. In this series of articles I hope to shed some light on the dark corners of web services and help navigate the sea of alphabet soup ([1](http://www.ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest" \l "fn1)). Along the way I'll also cover some tools for developing web services, and create a simple Web Service as an example. In this article I will give a high-level overview of both SOAP and REST.

**Introduction**

There are currently two schools of thought in developing web services: the traditional, standards-based approach (SOAP) and conceptually simpler and the trendier new kid on the block (REST). The decision between the two will be your first choice in designing a web service, so it is important to understand the pros and cons of the two. It is also important, in the sometimes heated debate between the two philosophies, to separate reality from rhetoric.

**SOAP**

In the beginning there was...SOAP. Developed at Microsoft in 1998, the inappropriately-named "Simple Object Access Protocol" was designed to be a platform and language-neutral alternative to previous middleware techologies like [CORBA](http://www.omg.org/docs/formal/04-03-12.pdf) and DCOM. Its first public appearance was an Internet public draft (submitted to the IETF) in 1999; shortly thereafter, in December of 1999, SOAP 1.0 was released. In May of 2000 the 1.1 version was submitted to the [W3C](http://www.w3.org/TR/soap/) where it formed the heart of the emerging Web Services technologies. The current version is 1.2, finalized in 2005. The examples given in this article will all be SOAP 1.2.

Together with WSDL and XML Schema, SOAP has become the standard for exchanging XML-based messages. SOAP was also designed from the ground up to be extensible, so that other standards could be integrated into it--and there have been many, often collectively referred to as WS-\*: WS-Addressing, WS-Policy, WS-Security, WS-Federation, WS-ReliableMessaging, WS-Coordination, WS-AtomicTransaction, WS-RemotePortlets, and the [list goes on](http://dev2dev.bea.com/webservices/standards.html). Hence much of the perceived complexity of SOAP, as in Java, comes from the multitude of standards which have evolved around it. This should not be reason to be too concerned: as with other things, you only have to use what you actually need.

The basic structure of SOAP is like any other message format (including HTML itself): header and body. In SOAP 1.2 this would look something like

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">

<env:Header>

<!-- Header information here -->

</env:Header>

<env:Body>

<!-- Body or "Payload" here, a Fault if error happened -->

</env:Body>

</env:Envelope>

Note that the <Header> element is optional here, but the <Body> is mandatory.

**The SOAP <Header>**

SOAP uses special attributes in the standard "soap-envelope" namespace to handle the extensibility elements that can be defined in the header. The most important of these is the mustUnderstand attribute. By default, any element in the header can be safely ignored by the SOAP message recipient unless the the mustUnderstand attribute on the element is set to "true" (or "1", which is the only value recognized in SOAP 1.1). A good example of this would be a security token element that authenticates the sender/requestor of the message. If for some reason the recipient is not able to process these elements, a fault should be delivered back to the sender with a fault code of MustUnderstand.

Because SOAP is designed to be used in a network environment with multiple intermediaries (SOAP "nodes" as identified by the <Node> element), it also defines the special XML attributes role to manage which intermediary should process a given header element and relay, which is used to indicate that this element should be passed to the next node if not processed in the current one.

**The SOAP <Body>**

The SOAP body contains the "payload" of the message, which is defined by the WSDL's <Message> part. If there is an error that needs to be transmitted back to the sender, a single <Fault> element is used as a child of the <Body>.

**The SOAP <Fault>**

The <Fault> is the standard element for error handling. When present, it is the only child element of the SOAP <Body>. The structure of a fault looks like:

<env:Fault xmlns:m="http://www.example.org/timeouts">

<env:Code>

<env:Value>env:Sender</env:Value>

<env:Subcode>

<env:Value>m:MessageTimeout</env:Value>

</env:Subcode>

</env:Code>

<env:Reason>

<env:Text xml:lang="en">Sender Timeout</env:Text>

</env:Reason>

<env:Detail>

<m:MaxTime>P5M</m:MaxTime>

</env:Detail>

</env:Fault>

Here, only the <Code> and <Reason> child elements are required, and the <Subcode> child of <Code> is also optional. The body of the Code/Value element is a fixed enumeration with the values:

* VersionMismatch: this indicates that the node that "threw" the fault found an invalid element in the SOAP envelope, either an incorrect namespace, incorrect local name, or both.
* MustUnderstand: as discussed above, this code indicates that a header element with the attribute mustUnderstand="true" could not be processed by the node throwing the fault. A NotUnderstood header block should be provided to detail all of the elements in the original message which were not understood.
* DataEncodingUnknown: the data encoding specified in the envelope's encodingSytle attribute is not supported by the node throwing the fault.
* Sender: This is a "catch-all" code indicating that the message sent was not correctly formed or did not have the appropriate information to succeed.
* Receiver: Another "catch-all" code indicating that the message could not be processed for reasons attributable to the processing of the message rather than to the contents of the message itself.

Subcodes, however, are not restricted and are application-defined; these will commonly be defined when the fault code is Sender or Receiver. The <Reason> element is there to provide a human-readable explanation of the fault. The optional <Detail> element is there to provide additional information about the fault, such as (in the example above) the timeout value. <Fault> also has optional children <Node> and <Role>, indicating which node threw the fault and the role that the node was operating in (see role attribute above) respectively.

**SOAP Encoding**

[Section 5](http://www.w3.org/TR/2000/NOTE-SOAP-20000508/" \l "_Toc478383512) of the SOAP 1.1 specification describes SOAP encoding, which was originally developed as a convenience for serializing and de-serializing data types to and from other sources, such as databases and programming languages. Problems, however, soon arose with [complications](http://msdn.microsoft.com/en-us/library/ms995710.aspx) in reconciling SOAP encoding and XML Schema, as well as with [performance](http://www.ibm.com/developerworks/webservices/library/ws-soapenc/). The WS-I organization finally put the nail in the coffin of SOAP encoding in 2004 when it released the first version of the [WS-I Basic Profile](http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html), declaring that only literal XML messages should be used (R2706). With the wide acceptance of WS-I, some of the more recent web service toolkits do not provide any support for (the previously ubiquitous) SOAP encoding at all.

**A Simple SOAP Example**

Putting it all together, below is an example of a simple request-response in SOAP for a stock quote. Here the transport binding is HTTP.

The request:

GET /StockPrice HTTP/1.1

Host: example.org

Content-Type: application/soap+xml; charset=utf-8

Content-Length: nnn

<?xml version="1.0"?>

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"

xmlns:s="http://www.example.org/stock-service">

<env:Body>

<s:GetStockQuote>

<s:TickerSymbol>IBM</s:TickerSymbol>

</s:GetStockQuote>

</env:Body>

</env:Envelope>

The response:

HTTP/1.1 200 OK

Content-Type: application/soap+xml; charset=utf-8

Content-Length: nnn

<?xml version="1.0"?>

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"

xmlns:s="http://www.example.org/stock-service">

<env:Body>

<s:GetStockQuoteResponse>

<s:StockPrice>45.25</s:StockPrice>

</s:GetStockQuoteResponse>

</env:Body>

</env:Envelope>

If you play your cards right, you may never have to actually see a SOAP message in action; every SOAP engine out there will do its best to hide it from you unless you really want to see it. If something goes wrong in your web service, however, it may be useful to know what one looks like for debugging purposes.

**REST**

Much in the way that [Ruby on Rails](http://www.rubyonrails.org/) was a reaction to more complex web application architectures, the emergence of the RESTful style of web services was a reaction to the more heavy-weight SOAP-based standards. In RESTful web services, the emphasis is on simple point-to-point communication over HTTP using plain old XML (POX).

The origin of the term "REST" comes from the famous [thesis](http://www.ics.uci.edu/%7Efielding/pubs/dissertation/top.htm) from Roy Fielding describing the concept of Representative State Transfer (REST). REST is an architectural style that can be summed up as four verbs (GET, POST, PUT, and DELETE from HTTP 1.1) and the nouns, which are the resources available on the network (referenced in the URI). The verbs have the following operational equivalents:

HTTP CRUD Equivalent

==============================

GET read

POST create,update,delete

PUT create,update

DELETE delete

A service to get the details of a user called 'dsmith', for example, would be handled using an HTTP GET to http://example.org/users/dsmith. Deleting the user would use an HTTP DELETE, and creating a new one would mostly likely be done with a POST. The need to reference other resources would be handled using hyperlinks (the XML equivalent of HTTP's href, which is [XLinks'](http://www.w3.org/TR/xlink/) xlink:href) and separate HTTP request-responses.

**A Simple RESTful Service**

Re-writing the stock quote service above as a RESTful web service provides a nice illustration of the differences between SOAP and REST web services.

The request:

GET /StockPrice/IBM HTTP/1.1

Host: example.org

Accept: text/xml

Accept-Charset: utf-8

The response:

HTTP/1.1 200 OK

Content-Type: text/xml; charset=utf-8

Content-Length: nnn

<?xml version="1.0"?>

<s:Quote xmlns:s="http://example.org/stock-service">

<s:TickerSymbol>IBM</s:TickerSymbol>

<s:StockPrice>45.25</s:StockPrice>

</s:Quote>

Though slightly modified (to include the ticker symbol in the response), the RESTful version is still simpler and more concise than the RPC-style SOAP version. In a sense, as well, RESTful web services are much closer in design and philosophy to the Web itself.

**Defining the Contract**

Traditionally, the big drawback of REST vis-a-vis SOAP was the lack of any way of specifying a description/contract for the web service. This, however, has changed since WSDL 2.0 defines a full compliment of non-SOAP bindings (all the HTTP methods, not just GET and POST) and the emergence of WADL as an alternative to WSDL. This will be discussed in more detail in coming articles.

**Summary and Pros/Cons**

SOAP and RESTful web services have a very different philosophy from each other. SOAP is really a protocol for XML-based distributed computing, whereas REST adheres much more closely to a bare metal, web-based design. SOAP by itself is not that complex; it can get complex, however, when it is used with its numerous extensions (guilt by association).

To summarize their strengths and weaknesses:

\*\*\* SOAP \*\*\*

Pros:

* Langauge, platform, and transport agnostic
* Designed to handle distributed computing environments
* Is the prevailing standard for web services, and hence has better support from other standards (WSDL, WS-\*) and tooling from vendors
* Built-in error handling (faults)
* Extensibility

Cons:

* Conceptually more difficult, more "heavy-weight" than REST
* More verbose
* Harder to develop, requires tools

\*\*\* REST \*\*\*

Pros:

* Language and platform agnostic
* Much simpler to develop than SOAP
* Small learning curve, less reliance on tools
* Concise, no need for additional messaging layer
* Closer in design and philosophy to the Web

Cons:

* Assumes a point-to-point communication model--not usable for distributed computing environment where message may go through one or more intermediaries
* Lack of standards support for security, policy, reliable messaging, etc., so services that have more sophisticated requirements are harder to develop ("roll your own")
* Tied to the HTTP transport model

|  |  |  |  |
| --- | --- | --- | --- |
| **Criteria** | **SOAP** | **RESTful** | **Comments** |
| Specification | JAX-WS | JAX-RS | Added on 28th Dec 2011. |
| Orientation | Wraps business logic | Accesses resources/data |  |
| Developer View | Object oriented | Resource Oriented |  |
| Language Independence | Yes | Yes |  |
| Platform Independence | Yes | Yes |  |
| Simplicity | No | Yes |  |
| Standards Based | Yes | No | SOAP web services are based on SOAP and WS-\* specifications |
| Security | SSL, WS-Security | SSL | WS-Security provides end-to-end security covering message integrity and authentication |
| Transactions | WS-AtomicTransaction | No |  |
| Reliability | WS-ReliableMessaging | Application specific |  |
| Performance | Good | Better | Caching and lower message payload makes RESTful web services performance efficient and scalable |
| Caching | No | GET operations can be cached |  |
| Transport protocol support | HTTP, SMTP, JMS | HTTP | Multiple transport protocol support makes SOAP Web Services flexible |
| Message Size | Heavy, has SOAP and WS-\* specific markup | Lightweight, no extra xml markup |  |
| Message Communication protocol | XML | XML, JSON, other valid MIME type | This flexibility of REST makes its extremely useful in providing consumer need specific message payloads |
| Message Encoding | Yes | No | SOAP Web Services support text and binary encoding, RESTful encoding is limited to text |
| Service Description | WSDL | No formal contract definition | In REST, no formal way to describe a service interface means more dependence on written documentation |
| Human intelligible Payload | No | Yes |  |
| Developer Tooling | Yes | Minimal or none | Complexity of SOAP Web Services dictates the need for using frameworks to facilitate rapid application development. REST on the other hand due to its simplicity can be developed without any framework |
| Attachment Support | via SAAJ specification | RESTful WS implementation specific support | Added on 28th Dec 2011. Thanks [Ron Grimes](http://technicalmumbojumbo.wordpress.com/2011/01/15/interview-question-soap-restful-webservices-comparison-soap-vs-restful/#comment-777) |

Now we come to most important stage of deciding which type to select. There is no one answer for selecting either SOAP based or RESTful Web Services. Neither is the right choice for every situation. The choice is dependant upon specific needs and which solution addresses them best. An architect should ask the following questions and the responses should assist him/her in making an informed decision:

* Does the service expose data or business logic? (REST can be a good choice for exposing data, SOAP/WS-\* might be a better choice for logic)
* Does the service need the capabilities of WS-\*, or is a simpler RESTful approach sufficient?
* What’s best for the developers who will build clients for the service?

Areas where RESTful WebServices are a great choice:

* **Limited bandwidth and resources:** Remember the return structure is really in any format (developer defined). Plus, any browser can be used because the REST approach uses the standard GET, PUT, POST, and DELETE verbs. Again, remember that REST can also use the XMLHttpRequest object that most modern browsers support today, which adds an extra bonus of AJAX.
* **Totally stateless operations:** If an operation needs to be continued, then REST is not the best approach and SOAP may fit it better. However, if you need stateless CRUD (Create, Read, Update, and Delete) operations, then REST is suitable.
* **Caching situations:** If the information can be cached because of the totally stateless operation of the REST approach, this is perfect.

Areas where SOAP based WebServices is a great solution:

* **Asynchronous processing and invocation:** If application needs a guaranteed level of reliability and security then SOAP 1.2 offers additional standards to ensure this type of operation. Things like WSRM – WS-Reliable Messaging etc.
* **Formal contracts:** If both sides (provider and consumer) have to agree on the exchange format then SOAP 1.2 gives the rigid specifications for this type of interaction.
* **Stateful operations:** If the application needs contextual information and conversational state management then SOAP 1.2 has the additional specification in the WS\* structure to support those things (Security, Transactions, Coordination, etc). Comparatively, the REST approach would make the developers build this custom plumbing.