**REST and SOAP: When Should I Use Each (or Both)?**

**By:**[**Michael Findling**](http://gp.embarcadero.com/authors/edit/14137.aspx)

**Web developers today have a myriad of technologies they can choose from; everything from simplified database access, to easy wrapping of existing middleware services, to a plethora of interesting client side software. All of these products and tools are there to give web developers the ability to create the best web-based applications in the shortest amount of time.**

However, having a massive set of possible software solutions is one challenge, picking the specific approach for specific parts of the web applications is another, and web developers today have to juggle many of these decisions with changing standards or approaches seemingly appearing daily.

Take for example, the two approaches for interfacing to the web with web services, namely SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). Both approaches work, both have advantages and disadvantages to interfacing to web services, but it is up to the web developer to make the decision of which approach may be best for each particular case.

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.

As shown above, each technology approach has their uses. They both have underlying issues around security, transport layers, and the like, but they both can get the job done and in many cases, they each bring something to the web. So for this argument, the best rule, is the rule of flexibility, because no matter what the problem at least in today’s web development world, web developers have great solutions using either of these protocols.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Stackoverflow.com\*\*\*\*\*\*\*\*

Unfortunately, there are a lot of misinformation and misconceptions around REST.

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, but it's easier to ascertain if the contract is being followed.

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 it 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.

Following 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 a standardized URI scheme.
* REST is not a mapping of CRUD to HTTP methods. Read [this](https://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 are standardized, so that's what you use when doing REST over HTTP.
* REST is not REST without [hypermedia](https://stackoverflow.com/a/29586455/1202421) and [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS). This means that 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 a 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 mistakenly called 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.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Restful**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**REpresentational State Transfer** (**REST**), or **RESTful,** [web services](https://en.wikipedia.org/wiki/Web_service) provide interoperability between computer systems on the [Internet](https://en.wikipedia.org/wiki/Internet). REST-compliant web services allow the requesting systems to access and manipulate textual representations of [web resources](https://en.wikipedia.org/wiki/Web_resource) by using a uniform and predefined set of [stateless](https://en.wikipedia.org/wiki/Stateless_protocol) operations. Other kinds of web services, such as [WSDL](https://en.wikipedia.org/wiki/Web_Services_Description_Language) and [SOAP](https://en.wikipedia.org/wiki/SOAP), expose their own arbitrary sets of operations.[[1]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-1)

"Web resources" were first defined on the [World Wide Web](https://en.wikipedia.org/wiki/World_Wide_Web) as documents or files identified by their [URLs](https://en.wikipedia.org/wiki/URL). However, today they have a much more generic and abstract definition that encompasses every thing or entity that can be identified, named, addressed, or handled, in any way whatsoever, on the web. In a RESTful web service, requests made to a resource's [URI](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier) will elicit a response that may be in [XML](https://en.wikipedia.org/wiki/XML), [HTML](https://en.wikipedia.org/wiki/HTML), [JSON](https://en.wikipedia.org/wiki/JSON), or some other format. The response may confirm that some alteration has been made to the stored resource, and the response may provide [hypertext](https://en.wikipedia.org/wiki/Hypertext) links to other related resources or collections of resources. When [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) is used, as is most common, the operations available are GET, POST, PUT, DELETE, and other predefined [CRUD](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) [HTTP methods](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods).

## Architectural properties

The constraints of the REST architectural style affect the following architectural properties:[[2]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-2)[[8]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-SOA_with_REST-8)

* Performance - component interactions can be the dominant factor in user-perceived performance and network efficiency[[9]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch2-9)
* [Scalability](https://en.wikipedia.org/wiki/Scalability) to support large numbers of components and interactions among components. Roy Fielding, one of the principal authors of the HTTP specification, describes REST's effect on scalability as follows:

REST's client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—[proxies](https://en.wikipedia.org/wiki/Proxy_server), [gateways](https://en.wikipedia.org/wiki/Gateway_(telecommunications)), and [firewalls](https://en.wikipedia.org/wiki/Firewall_(computing))—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate [cacheability](https://en.wikipedia.org/wiki/Web_cache" \o "Web cache).

* Simplicity of a uniform Interface
* Modifiability of components to meet changing needs (even while the application is running)
* Visibility of communication between components by service agents
* Portability of components by moving program code with the data
* [Reliability](https://en.wiktionary.org/wiki/reliability) is the resistance to failure at the system level in the presence of failures within components, connectors, or data

## Architectural constraints

Six guiding constraints define a RESTful system. These constraints restrict the ways that the server may process and respond to client requests so that, by operating within these constraints, the service gains desirable [non-functional properties](https://en.wikipedia.org/wiki/Non-functional_requirement), such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[[2]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-2) If a service violates any of the required constraints, it cannot be considered RESTful.

The formal REST constraints are as follows:

### Client-server architecture

*See also:*[*Client–server model*](https://en.wikipedia.org/wiki/Client%E2%80%93server_model)

The principle behind the client-server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.[[2]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-2)

### Statelessness

*See also:*[*Stateless protocol*](https://en.wikipedia.org/wiki/Stateless_protocol)

The client–server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.

### Cacheability

*See also:*[*Web cache*](https://en.wikipedia.org/wiki/Web_cache)

As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

### Layered system

*See also:*[*Layered system*](https://en.wikipedia.org/wiki/Layered_system)

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.

### Code on demand (optional)

*See also:*[*Client-side scripting*](https://en.wikipedia.org/wiki/Client-side_scripting)

Servers can temporarily extend or customize the functionality of a client by transferring executable code. Examples of this may include compiled components such as [Java applets](https://en.wikipedia.org/wiki/Java_applet) and client-side scripts such as [JavaScript](https://en.wikipedia.org/wiki/JavaScript).

### Uniform interface

The uniform interface constraint is fundamental to the design of any REST service.[[2]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-2) It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

#### Resource identification in requests

Individual resources are identified in requests, for example using [URIs](https://en.wikipedia.org/wiki/Uniform_resource_identifier) in Web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as [HTML](https://en.wikipedia.org/wiki/HTML), [XML](https://en.wikipedia.org/wiki/XML) or [JSON](https://en.wikipedia.org/wiki/JSON), none of which are the server's internal representation.

#### Resource manipulation through representations

When a client holds a representation of a resource, including any [metadata](https://en.wikipedia.org/wiki/Metadata) attached, it has enough information to modify or delete the resource.

#### Self-descriptive messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an [Internet media type](https://en.wikipedia.org/wiki/Media_type) (previously known as a [MIME type](https://en.wikipedia.org/wiki/MIME_type)).[[2]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-2)

#### Hypermedia as the engine of application state ([HATEOAS](https://en.wikipedia.org/wiki/HATEOAS))

Having accessed an initial URI for the REST application—analogous to a human Web user accessing the [home page](https://en.wikipedia.org/wiki/Home_page) of a website—a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes [hyperlinks](https://en.wikipedia.org/wiki/Hyperlink) to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the REST service.

## Applied to Web services

Web service [APIs](https://en.wikipedia.org/wiki/Application_programming_interface) that adhere to the [REST architectural constraints](https://en.wikipedia.org/wiki/Representational_state_transfer#Architectural_constraints) are called RESTful APIs. HTTP-based RESTful APIs are defined with the following aspects:

* base [URL](https://en.wikipedia.org/wiki/URL), such as http://api.example.com/resources
* an [internet media type](https://en.wikipedia.org/wiki/Internet_media_type) that defines state transition data elements (e.g., Atom, microformats, application/vnd.collection+json etc.) The current representation tells the client how to compose requests for transitions to all the next available application states. This could be as simple as a URL or as complex as a Java applet.[[15]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-must_be_hypertext_driven-15)
* standard [HTTP methods](https://en.wikipedia.org/wiki/HTTP_method) (e.g., OPTIONS, GET, PUT, POST, and DELETE)[[16]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-16)

### Relationship between URL and HTTP methods[[edit](https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&action=edit&section=15)]

The following table shows how HTTP methods are typically used in a RESTful API:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **HTTP methods** | | | | | |
| **Uniform Resource Locator (URL)** | **GET** | **PUT** | **PATCH** | **POST** | **DELETE** |
| **Collection, such as**  **https://api.example.com/resources/** | **List** the URIs and perhaps other details of the collection's members. | **Replace**  the entire collection with another collection. | Not generally used | **Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation. | **Delete** the entire collection. |
| **Element, such as**  **https://api.example.com/resources/item17** | **Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type. | **Replace** the addressed member of the collection, or if it does not exist, **create**it. | **Update** the addressed member of the collection. | Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry within it.[[17]](https://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-thereisnorightway-17) | **Delete** the addressed member of the collection. |

The GET method is a [safe method](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Safe_methods) (or *[nullipotent](https://en.wiktionary.org/wiki/nullipotent" \o "wiktionary:nullipotent)*), meaning that calling it produces no [side-effects](https://en.wikipedia.org/wiki/Side_effect_(computer_science)): retrieving or accessing a record does not change it. The PUT and DELETE methods are [idempotent](https://en.wikipedia.org/wiki/Idempotent#Computer_science_meaning), meaning that the state of the system exposed by the API is unchanged no matter how many times more than once the same request is repeated.

Unlike [SOAP](https://en.wikipedia.org/wiki/SOAP)-based Web services, there is no "official" standard for RESTful Web APIs. This is because REST is an architectural style, while SOAP is a protocol. REST is not a standard in itself, but RESTful implementations make use of standards, such as [HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol), [URI](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier), [JSON](https://en.wikipedia.org/wiki/JSON), and [XML](https://en.wikipedia.org/wiki/XML). Many developers also describe their APIs as being RESTful, even though these APIs actually don't fulfill all of the architectural constraints described above (especially the uniform interface constraint).