**What Are RESTful Web Services?**

**RESTful web services** are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

* **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See [The @Path Annotation and URI Path Templates](http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#ginpw) for more information.
* **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See [Responding to HTTP Methods and Requests](http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipys) for more information.
* **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See [Responding to HTTP Methods and Requests](http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipys) and [Using Entity Providers to Map HTTP Response and Request Entity Bodies](http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipze) for more information.
* **Stateful interactions through hyperlinks**: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See [Using Entity Providers to Map HTTP Response and Request Entity Bodies](http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html#gipze) and “Building URIs” in the JAX-RS Overview document for more information.

**Architectural constraints**

The architectural properties of REST are realized by applying specific interaction constraints to components, connectors, and data elements.The formal REST constraints are:

**Client–server**

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

A uniform interface separates clients from servers. This [separation of concerns](http://en.wikipedia.org/wiki/Separation_of_concerns) means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the [portability](http://en.wikipedia.org/wiki/Software_portability) of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more [scalable](http://en.wikipedia.org/wiki/Scalability). Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

**Stateless**

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

The client–server communication is further 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.

**Cacheable**

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

As on the World Wide Web, clients 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**

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.

**Uniform interface**

The uniform interface constraint is fundamental to the design of any REST service.[[1]](http://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-1) The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

**Identification of resources**

Individual resources are identified in requests, for example using [URIs](http://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](http://en.wikipedia.org/wiki/HTML), [XML](http://en.wikipedia.org/wiki/XML) or [JSON](http://en.wikipedia.org/wiki/JSON), none of which are the server's internal representation, and it is the same one resource regardless.

**Manipulation of resources through these representations**

When a client holds a representation of a resource, including any [metadata](http://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](http://en.wikipedia.org/wiki/Internet_media_type)(previously known as a [MIME](http://en.wikipedia.org/wiki/MIME) type). Responses also explicitly indicate their cacheability.[[1]](http://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Fielding-Ch5-1)

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

Clients make state transitions only through actions that are dynamically identified within hypermedia by the server (e.g., by [hyperlinks](http://en.wikipedia.org/wiki/Hyperlink) within [hypertext](http://en.wikipedia.org/wiki/Hypertext)). Except for simple fixed entry points to the application, a client does not assume that any particular action is available for any particular resources beyond those described in representations previously received from the server.

One can characterise applications conforming to the REST constraints described in this section as "RESTful".[[4]](http://en.wikipedia.org/wiki/Representational_state_transfer#cite_note-Richardson_2007-4) If a service violates any of the required constraints, it cannot be considered RESTful.

Complying with these constraints, and thus conforming to the REST architectural style, enables any kind of distributed hypermedia system to have desirable [emergent properties](http://en.wikipedia.org/wiki/Emergence), such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

Unlike [SOAP](http://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, unlike SOAP, which is a protocol. Even though REST is not a standard per se, most RESTful implementations make use of standards like [HTTP](http://en.wikipedia.org/wiki/HTTP), [URI](http://en.wikipedia.org/wiki/URI), [JSON](http://en.wikipedia.org/wiki/JSON), [XML](http://en.wikipedia.org/wiki/XML), etc.

**Richardson Maturity Model**

**steps toward the glory of REST**

*A model (developed by Leonard Richardson) that breaks down the principal elements of a REST approach into three steps. These introduce resources, http verbs, and hypermedia controls.*

Recently I've been reading drafts of [Rest In Practice](http://www.amazon.com/gp/product/0596805829?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0596805829)http://www.assoc-amazon.com/e/ir?t=martinfowlerc-20&l=as2&o=1&a=0321601912: a book that a couple of my colleagues have been working on. Their aim is to explain how to use Restful web services to handle many of the integration problems that enterprises face. At the heart of the book is the notion that the web is an existence proof of a massively scalable distributed system that works really well, and we can take ideas from that to build integrated systems more easily.

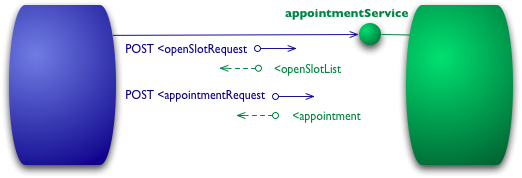


*Figure 1: Steps toward REST*

To help explain the specific properties of a web-style system, the authors use a model of restful maturity that was developed by [Leonard Richardson](http://www.crummy.com/) and [explained](http://www.crummy.com/writing/speaking/2008-QCon/act3.html) at a QCon talk. The model is nice way to think about using these techniques, so I thought I'd take a stab of my own explanation of it. (The protocol examples here are only illustrative, I didn't feel it was worthwhile to code and test them up, so there may be problems in the detail.)

## Level 0

The starting point for the model is using HTTP as a transport system for remote interactions, but without using any of the mechanisms of the web. Essentially what you are doing here is using HTTP as a tunneling mechanism for your own remote interaction mechanism, usually based on [Remote Procedure Invocation](http://www.eaipatterns.com/EncapsulatedSynchronousIntegration.html).



*Figure 2: An example interaction at Level 0*

Let's assume I want to book an appointment with my doctor. My appointment software first needs to know what open slots my doctor has on a given date, so it makes a request of the hospital appointment system to obtain that information. In a level 0 scenario, the hospital will expose a service endpoint at some URI. I then post to that endpoint a document containing the details of my request.

POST /appointmentService HTTP/1.1

[various other headers]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>

The server then will return a document giving me this information

HTTP/1.1 200 OK

[various headers]

<openSlotList>

<slot start = "1400" end = "1450">

<doctor id = "mjones"/>

</slot>

<slot start = "1600" end = "1650">

<doctor id = "mjones"/>

</slot>

</openSlotList>

I'm using XML here for the example, but the content can actually be anything: JSON, YAML, key-value pairs, or any custom format.

My next step is to book an appointment, which I can again do by posting a document to the endpoint.

POST /appointmentService HTTP/1.1

[various other headers]

<appointmentRequest>

<slot doctor = "mjones" start = "1400" end = "1450"/>

<patient id = "jsmith"/>

</appointmentRequest>

If all is well I get a response saying my appointment is booked.

HTTP/1.1 200 OK

[various headers]

<appointment>

<slot doctor = "mjones" start = "1400" end = "1450"/>

<patient id = "jsmith"/>

</appointment>

If there is a problem, say someone else got in before me, then I'll get some kind of error message in the reply body.

HTTP/1.1 200 OK

[various headers]

<appointmentRequestFailure>

<slot doctor = "mjones" start = "1400" end = "1450"/>

<patient id = "jsmith"/>

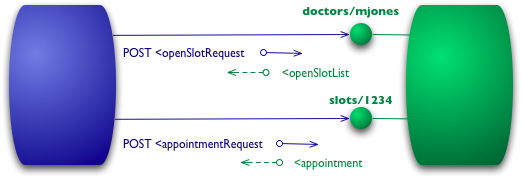
<reason>Slot not available</reason>

</appointmentRequestFailure>

So far this is a straightforward RPC style system. It's simple as it's just slinging plain old XML (POX) back and forth. If you use SOAP or XML-RPC it's basically the same mechanism, the only difference is that you wrap the XML messages in some kind of envelope.

## Level 1 - Resources

The first step towards the Glory of Rest in the RMM is to introduce resources. So now rather than making all our requests to a singular service endpoint, we now start talking to individual resources.



*Figure 3: Level 1 adds resources*

So with our initial query, we might have a resource for given doctor.

POST /doctors/mjones HTTP/1.1

[various other headers]

<openSlotRequest date = "2010-01-04"/>

The reply carries the same basic information, but each slot is now a resource that can be addressed individually.

HTTP/1.1 200 OK

[various headers]

<openSlotList>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>

<slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>

</openSlotList>

With specific resources booking an appointment means posting to a particular slot.

POST /slots/1234 HTTP/1.1

[various other headers]

<appointmentRequest>

<patient id = "jsmith"/>

</appointmentRequest>

If all goes well I get a similar reply to before.

HTTP/1.1 200 OK

[various headers]

<appointment>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>

<patient id = "jsmith"/>

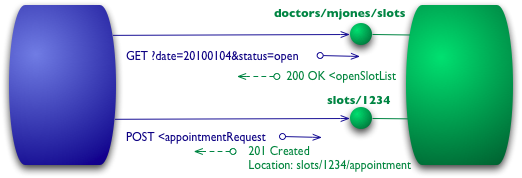
</appointment>

The difference now is that if anyone needs to do anything about the appointment, like book some tests, they first get hold of the appointment resource, which might have a URI like http://royalhope.nhs.uk/slots/1234/appointment, and post to that resource.

To an object guy like me this is like the notion of object identity. Rather than calling some function in the ether and passing arguments, we call a method on one particular object providing arguments for the other information.

## Level 2 - HTTP Verbs

I've used HTTP POST verbs for all my interactions here in level 0 and 1, but some people use GETs instead or in addition. At these levels it doesn't make much difference, they are both being used as tunneling mechanisms allowing you to tunnel your interactions through HTTP. Level 2 moves away from this, using the HTTP verbs as closely as possible to how they are used in HTTP itself.



*Figure 4: Level 2 addes HTTP verbs*

For our the list of slots, this means we want to use GET.

GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1

Host: royalhope.nhs.uk

The reply is the same as it would have been with the POST

HTTP/1.1 200 OK

[various headers]

<openSlotList>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>

<slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>

</openSlotList>

At Level 2, the use of GET for a request like this is crucial. HTTP defines GET as a safe operation, that is it doesn't make any significant changes to the state of anything. This allows us to invoke GETs safely any number of times in any order and get the same results each time. An important consequence of this is that it allows any participant in the routing of requests to use caching, which is a key element in making the web perform as well as it does. HTTP includes various measures to support caching, which can be used by all participants in the communication. By following the rules of HTTP we're able to take advantage of that capability.

To book an appointment we need an HTTP verb that does change state, a POST or a PUT. I'll use the same POST that I did earlier.

POST /slots/1234 HTTP/1.1

[various other headers]

<appointmentRequest>

<patient id = "jsmith"/>

</appointmentRequest>

The trade-offs between using POST and PUT here are more than I want to go into here, maybe I'll do a separate article on them some day. But I do want to point out that some people incorrectly make a correspondence between POST/PUT and create/update. The choice between them is rather different to that.

Even if I use the same post as level 1, there's another significant difference in how the remote service responds. If all goes well, the service replies with a response code of 201 to indicate that there's a new resource in the world.

HTTP/1.1 201 Created

Location: slots/1234/appointment

[various headers]

<appointment>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>

<patient id = "jsmith"/>

</appointment>

The 201 response includes a location attribute with a URI that the client can use to GET the current state of that resource in the future. The response here also includes a representation of that resource to save the client an extra call right now.

There is another difference if something goes wrong, such as someone else booking the session.

HTTP/1.1 409 Conflict

[various headers]

<openSlotList>

<slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>

</openSlotList>

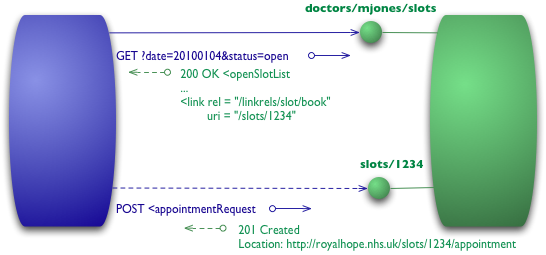
The important part of this response is the use of an HTTP response code to indicate something has gone wrong. In this case a 409 seems a good choice to indicate that someone else has already updated the resource in an incompatible way. Rather than using a return code of 200 but including an error response, at level 2 we explicitly use some kind of error response like this. It's up to the protocol designer to decide what codes to use, but there should be a non-2xx response if an error crops up. Level 2 introduces using HTTP verbs and HTTP response codes.

There is an inconsistency creeping in here. REST advocates talk about using all the HTTP verbs. They also justify their approach by saying that REST is attempting to learn from the practical success of the web. But the world-wide web doesn't use PUT or DELETE much in practice. There are sensible reasons for using PUT and DELETE more, but the existence proof of the web isn't one of them.

The key elements that are supported by the existence of the web are the strong separation between safe (eg GET) and non-safe operations, together with using status codes to help communicate the kinds of errors you run into.

## Level 3 - Hypermedia Controls

The final level introduces something that you often hear referred to under the ugly acronym of HATEOAS (Hypertext As The Engine Of Application State). It addresses the question of how to get from a list open slots to knowing what to do to book an appointment.



*Figure 5: Level 3 adds hypermedia controls*

We begin with the same initial GET that we sent in level 2

GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1

Host: royalhope.nhs.uk

But the response has a new element

HTTP/1.1 200 OK

[various headers]

<openSlotList>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450">

<link rel = "/linkrels/slot/book"

uri = "/slots/1234"/>

</slot>

<slot id = "5678" doctor = "mjones" start = "1600" end = "1650">

<link rel = "/linkrels/slot/book"

uri = "/slots/5678"/>

</slot>

</openSlotList>

Each slot now has a link element which contains a URI to tell us how to book an appointment.

The point of hypermedia controls is that they tell us what we can do next, and the URI of the resource we need to manipulate to do it. Rather than us having to know where to post our appointment request, the hypermedia controls in the response tell us how to do it.

The POST would again copy that of level 2

POST /slots/1234 HTTP/1.1

[various other headers]

<appointmentRequest>

<patient id = "jsmith"/>

</appointmentRequest>

And the reply contains a number of hypermedia controls for different things to do next.

HTTP/1.1 201 Created

Location: http://royalhope.nhs.uk/slots/1234/appointment

[various headers]

<appointment>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>

<patient id = "jsmith"/>

<link rel = "/linkrels/appointment/cancel"

uri = "/slots/1234/appointment"/>

<link rel = "/linkrels/appointment/addTest"

uri = "/slots/1234/appointment/tests"/>

<link rel = "self"

uri = "/slots/1234/appointment"/>

<link rel = "/linkrels/appointment/changeTime"

uri = "/doctors/mjones/slots?date=20100104@status=open"/>

<link rel = "/linkrels/appointment/updateContactInfo"

uri = "/patients/jsmith/contactInfo"/>

<link rel = "/linkrels/help"

uri = "/help/appointment"/>

</appointment>

One obvious benefit of hypermedia controls is that it allows the server to change its URI scheme without breaking clients. As long as clients look up the "addTest" link URI then the server team can juggle all URIs other than the initial entry points.

A further benefit is that it helps client developers explore the protocol. The links give client developers a hint as to what may be possible next. It doesn't give all the information: both the "latest" and "cancel" controls point to the same URI - they need to figure out that one is a GET and the other a DELETE. But at least it gives them a starting point as to what to think about for more information and to look for a similar URI in the protocol documentation.

Similarly it allows the server team to advertise new capabilities by putting new links in the responses. If the client developers are keeping an eye out for unknown links these links can be a trigger for further exploration.

There's no absolute standard as to how to represent hypermedia controls. What I've done here is to use the current recommendations of the REST in Practice team, which is to follow ATOM ([RFC 4287](http://atompub.org/rfc4287.html)) I use a <link> element with a uri attribute for the target URI and a rel attribute for to describe the kind of relationship. A well known relationship (such as self for a reference to the element itself) is bare, any specific to that server is a fully qualified URI. ATOM states that the definition for well-known linkrels is the [Registry of Link Relations](http://www.iana.org/assignments/link-relations.html). As I write these are confined to what's done by ATOM, which is generally seen as a leader in level 3 restfulness.

## The Meaning of the Levels

I should stress that the RMM, while a good way to think about what the elements of REST, is not a definition of levels of REST itself. Roy Fielding has made it clear that [level 3 RMM is a pre-condition of REST](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven). Like many terms in software, REST gets lots of definitions, but since Roy Fielding coined the term, his definition should carry more weight than most.

What I find useful about this RMM is that it provides a good step by step way to understand the basic ideas behind restful thinking. As such I see it as tool to help us learn about the concepts and not something that should be used in some kind of assessment mechanism. I don't think we have enough examples yet to be really sure that the restful approach is the right way to integrate systems, I do think it's a very attractive approach and the one that I would recommend in most situations.

Talking about this with Ian Robinson, he stressed that something he found attractive about this model when Leonard Richardson first presented it was its relationship to common design techniques.

* Level 1 tackles the question of handling complexity by using divide and conquer, breaking a large service endpoint down into multiple resources.
* Level 2 introduces a standard set of verbs so that we handle similar situations in the same way, removing unnecessary variation.
* Level 3 introduces discoverability, providing a way of making a protocol more self-documenting.

The result is a model that helps us think about the kind of HTTP service we want to provide and frame the expectations of people looking to interact with it.

**SOAP vs. REST Challenges**

The term web API generally refers to both sides of computer systems communicating over a network: the API services offered by a server, as well as the API offered by the client such as a web browser. The server-side portion of the web API is a programmatic interface to a defined request-response message system, and is typically referred to as the Web Service. There are several design models for web services, but the two most dominant are SOAP and REST.

**SOAP**

SOAP – Simple Object Access Protocol – is probably the better known of the two models.

SOAP relies heavily on XML, and together with schemas, defines a very strongly typed messaging framework. Every operation the service provides is explicitly defined, along with the XML structure of the request and response for that operation. Each input parameter is similarly defined and bound to a type: for example an integer, a string, or some other complex object.

All of this is codified in the WSDL – Web Service Description (or Definition, in later versions) Language. The WSDL is often explained as a contract between the provider and the consumer of the service. In programming terms the WSDL can be thought of as a method signature for the web service.

**Example:**

A sample message exchange looks like the following.  
  
A request from the client:

POST http://www.stgregorioschurchdc.org/cgi/websvccal.cgi HTTP/1.1  
Accept-Encoding: gzip,deflate  
Content-Type: text/xml;charset=UTF-8  
SOAPAction: "http://www.stgregorioschurchdc.org/Calendar#easter\_date"  
Content-Length: 479  
Host: www.stgregorioschurchdc.org  
Connection: Keep-Alive  
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)  
<?xml version="1.0"?>  
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:cal="http://www.stgregorioschurchdc.org/Calendar">  
<soapenv:Header/>  
<soapenv:Body>  
   <cal:easter\_date soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
   <year xsi:type="xsd:short">2014</year>  
</cal:easter\_date>  
</soapenv:Body>  
</soapenv:Envelope>

The response from the service:

HTTP/1.1 200 OK  
Date: Fri, 22 Nov 2013 21:09:44 GMT  
Server: Apache/2.0.52 (Red Hat)  
SOAPServer: SOAP::Lite/Perl/0.52  
Content-Length: 566  
Connection: close  
Content-Type: text/xml; charset=utf-8  
<?xml version="1.0" encoding="UTF-8"?>  
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"  
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<SOAP-ENV:Body>  
   <namesp1:easter\_dateResponse  
xmlns:namesp1="http://www.stgregorioschurchdc.org/Calendar">     
<s-gensym3 xsi:type="xsd:string">2014/04/20</s-gensym3>  
</namesp1:easter\_dateResponse>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>

From this example we can see the message was sent over HTTP. SOAP is actually agnostic of the underlying transport protocol and can be sent over almost any protocol such as HTTP, SMTP, TCP, or JMS. As was already mentioned, the SOAP message itself must be XML-formatted. As is normal for any XML document, there must be one root element: the Envelope in this case. This contains two required elements: the Header and the Body. The rest of the elements in this message are described by the WSDL.

The accompanying WSDL that defines the above service looks like this (the details are not important, but the entire document is shown here for completeness):

<?xml version="1.0"?>  
<definitions xmlns:tns="http://www.stgregorioschurchdc.org/Calendar"  
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"  
xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
xmlns="http://schemas.xmlsoap.org/wsdl/"  
name="Calendar" targetNamespace="http://www.stgregorioschurchdc.org/Calendar">  
<message name="EasterDate">  
   <part name="year" type="xsd:short"/>  
</message>  
<message name="EasterDateResponse">  
   <part name="date" type="xsd:string"/>  
</message>  
<portType name="EasterDateSoapPort">  
   <operation name="easter\_date" parameterOrder="year">  
     <input message="tns:EasterDate"/>  
     <output message="tns:EasterDateResponse"/>  
   </operation>  
</portType>  
<binding name="EasterDateSoapBinding" type="tns:EasterDateSoapPort">  
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>  
   <operation name="easter\_date">  
     <soap:operation soapAction="http://www.stgregorioschurchdc.org/Calendar#easter\_date"/>  
     <input>  
      <soap:body use="encoded" namespace="http://www.stgregorioschurchdc.org/Calendar" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>  
     </input>  
     <output>  
       <soap:body use="encoded" namespace="http://www.stgregorioschurchdc.org/Calendar" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>  
     </output>  
   </operation>  
</binding>  
<service name="Calendar">  
   <port name="EasterDateSoapPort" binding="tns:EasterDateSoapBinding">  
     <soap:address location="http://www.stgregorioschurchdc.org/cgi/websvccal.cgi"/>  
   </port>  
</service>  
</definitions>

Notice that all the parts of the message body are described in this document. Also note that, even though this document is intended to be primarily read by a computer, it is still relatively easy for a person with some programming knowledge to follow.

**WSDL**

The WSDL defines every aspect of the SOAP message. It is even able to define whether any element or attribute is allowed to appear multiple times, if it is required or optional, and can even dictate a specific order the elements must appear in.

It is a common misconception that the WSDL is a requirement.

It is a common misconception that the WSDL is a requirement for a SOAP service. SOAP was designed before the WSDL, and therefore the WSDL is optional. Although arguably, it is significantly harder to interface with a web service that does not have a WSDL.

On the other hand, if a developer is asked to interface with an existing SOAP web service, he only needs to be given the WSDL, and there are tools that do service discovery - generate method stubs with appropriate parameters in almost any language from that WSDL. Many test tools on the market work in the same way - a tester provides a URL to a WSDL, and the tools generate all the calls with sample parameters for all the available methods.

**Critique**

While the WSDL may seem like a great thing at first – it is self documenting and contains almost the complete picture of everything that is required to integrate with a service – it can also become a burden. Remember, the WSDL is a contract between you (the provider of the service) and every single one of your customers (consumers of the service).

WSDL changes also means client changes.

If you want to make a change to your API, even something as small as adding an optional parameter, the WSDL must change. And WSDL changes also means client changes - all your consumers must recompile their client application against this new WSDL. This small change greatly increases the burden on the development teams (on both sides of the communication) as well as the test teams. For this reason, the WSDL is viewed as a version lock-in, and most providers are very resistant to updating their API.

Furthermore, while SOAP offers some interesting flexibility, such as the ability to be transmitted over any transport protocol, nobody has really taken advantage of most of these. Thanks to how the Internet evolved, everything that matters runs over HTTP. There are new advances, but most of these are being hampered by infrastructure routers refusing to route non-standard HTTP traffic. Just consider: how long has the world been trying to switch over to IPv6?

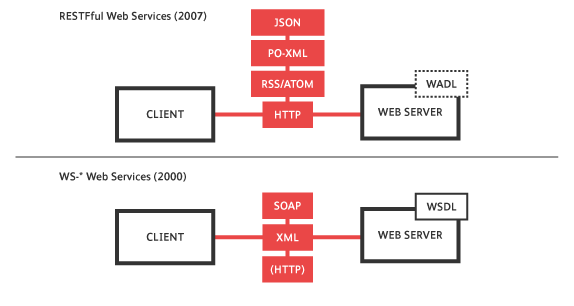
There is definitely a need for a more lightweight and flexible model [than SOAP].

Any situation where the size of the transmitted message does not matter, or where you control everything end-to-end, SOAP is almost always the better answer. This applies primarily to direct server to server communication, generally used for internal communication only within the confines of one company. However, there is a need for a world where almost every person on the planet has several low-memory, low-processing-power devices connected to multiple services at all times, there is definitely a need for a more lightweight and flexible model.

**REST**

REST – REpresentational State Transfer – is quickly becoming the preferred design model for public APIs. Some interesting statistics regarding the growth of REST in the public sphere are available at, for example, the [**Programmable Web**](http://www.programmableweb.com/apis).

REST is an architectural style, unlike SOAP which is a standardized protocol. REST makes use of existing and widely adopted technologies, specifically HTTP, and does not create any new standards. It can structure data into XML, [**YAML**](http://yaml.org/), or any other machine readable format, but usually JSON – JavaScript Object Notation – is preferred. As can be expected from JavaScript, the objects are not strongly typed. REST follows the object oriented programming paradigm of noun-verb. REST is very data-driven, compared to SOAP, which is strongly function-driven. In the REST paradigm, metadata is structured hierarchically and represented in the URI; this takes the place of the noun. The HTTP standard offers several verbs representing operations or actions you can perform on the data, most commonly: GET, POST, PUT, and DELETE.



*Source: http://dret.net/netdret/docs/soa-rest-www2009/rest-ws.pdf*

For example, let's say we need three operations: a user login, logout, and retrieve the current user's account balance. While a SOAP service would implement each of these as separate operations (with the username and password passed as arguments to the login operation), REST would define a URI like *http://sample.test/api/session*. A POST action (with username and password passed in the body) to this URI would accomplish a user being logged into the session, and a DELETE action to the same URI would accomplish the session being terminated – effectively a logout. A GET action to say*http://sample.test/api/session/balance* would retrieve the current user's balance.

**Example:**

A sample message exchange could contain as little as this.

**Request:**

GET http://www.catechizeme.com/catechisms/catechism\_for\_young\_children/daily\_question.js HTTP/1.1  
Accept-Encoding: gzip,deflate  
Host: www.catechizeme.com  
Connection: Keep-Alive  
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

**Response:**

HTTP/1.1 200 OK  
Date: Fri, 22 Nov 2013 22:32:22 GMT  
Server: Apache  
X-Powered-By: Phusion Passenger (mod\_rails/mod\_rack) 3.0.17  
ETag: "b8a7ef8b4b282a70d1b64ea5e79072df"  
X-Runtime: 13  
Cache-Control: private, max-age=0, must-revalidate  
Content-Length: 209  
Status: 200  
Keep-Alive: timeout=2, max=100  
Connection: Keep-Alive  
Content-Type: js; charset=utf-8  
{  
   "link": "catechisms\/catechism\_for\_young\_children\/questions\/36",  
   "catechism": "Catechism for Young Children",  
   "a": "Original sin.",  
   "position": 36,  
   "q": " What is that sinful nature which we inherit from Adam called?"  
}

As is already expected this message was sent over HTTP, and used the GET verb. Further note that the URI, which also had to be included in the SOAP request, but there it had no meaning, here actually takes on a meaning. The body of the message is significantly smaller, in this example there actually isn't one.

A REST service also has a schema in what is called a WADL – Web Application Description Language. The WADL for the above call would look like this:

<?xml version="1.0"?>  
<application xmlns="http://wadl.dev.java.net/2009/02">  
<doc xml:lang="en" title="http://www.catechizeme.com"/>  
<resources base="http://www.catechizeme.com">  
<resource path="catechisms/{CATECHISM\_NAME}/daily\_question.js" id="Daily\_question.js">  
<doc xml:lang="en" title="Daily\_question.js"/>  
<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="CATECHISM\_NAME" style="template" type="string"/>  
<method name="GET" id="Daily\_question.js">  
   <doc xml:lang="en" title="Daily\_question.js"/>  
   <request/>  
   <response status="200">  
     <representation mediaType="json" element="data"/>  
     <representation mediaType="js; charset=utf-8" element="data"/>  
   </response>  
</method>  
</resource>  
</resources>  
</application>

The WADL uses XML syntax to describe the metadata and the available actions. It can also be written to be as strict as the WSDL: defining types, optional parameters, etc.

**WADL**

The WADL does not have any mechanism to represent the data itself, which is what must be sent on the URI. This means that the WADL is able to document only about half of the information you need in order to interface with the service. Take for example the parameter CATECHISM\_NAME in the above sample. The WADL only tells you where in the URI the parameter belongs, and that it should be a string. However, if you had to glean the valid values for yourself, it would probably take you quite a long time. Note that it is possible to add a schema to the WADL, so that you can define even complex variable types such as enumerations; however, this is even more rare than providing a WADL.

The WADL is completely optional.

Further the WADL is completely optional; in fact, it is quite rare that the WADL is supplied at all! Due to the nature of the service, in order to make any meaningful use of it, you will almost undoubtedly need additional documentation.

**Critique**

Having a very small footprint and making use of the widely adopted HTTP standard makes REST a very attractive option for public APIs. Coupled together with JSON, which makes something like adding an optional parameter very simple, makes it very flexible and allows for frequent releases without impacting your consumers.

Arguably, the biggest drawback is the WADL – optional and lacking some necessary information. To address this deficiency, there are several frameworks available on the market that help document and produce RESTful APIs, such as [**Swagger**](https://developers.helloreverb.com/swagger/), [**RAML**](http://raml.org/), or [**JSON-home**](https://github.com/otto-de/jsonhome).

API designers, please note the following rules before calling your creation a REST API:

* A REST API should not be dependent on any single communication protocol, though its successful mapping to a given protocol may be dependent on the availability of metadata, choice of methods, etc. In general, any protocol element that uses a URI for identification must allow any URI scheme to be used for the sake of that identification. *[Failure here implies that identification is not separated from interaction.]*
* A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols, such as HTTP’s PATCH method or Link header field. Workarounds for broken implementations (such as those browsers stupid enough to believe that HTML defines HTTP’s method set) should be defined separately, or at least in appendices, with an expectation that the workaround will eventually be obsolete. *[Failure here implies that the resource interfaces are object-specific, not generic.]*
* A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). *[Failure here implies that out-of-band information is driving interaction instead of hypertext.]*
* A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations. *[Failure here implies that clients are assuming a resource structure due to out-of band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC's functional coupling].*
* A REST API should never have “typed” resources that are significant to the client. Specification authors may use resource types for describing server implementation behind the interface, but those types must be irrelevant and invisible to the client. The only types that are significant to a client are the current representation’s media type and standardized relation names.*[ditto]*
* A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user’s manipulation of those representations. The transitions may be determined (or limited by) the client’s knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand). *[Failure here implies that out-of-band information is driving interaction instead of hypertext.]*