Published on **XML.com** http://www.xml.com/pub/a/ws/2002/02/20/rest.html
See this if you're having trouble printing code examples

REST and the Real World

By Paul Prescod
February 20, 2002

In the last article I described a new model for web services construction. It is called Representational State Transfer (REST), and it applies the principles of the Web to transaction-oriented services, rather than publishing-oriented sites. When we apply the strategy in the real world, we do so using web technologies such as URIs, HTTP, and XML. Unlike the current generation of web services technologies, however, we make those three technologies central rather than peripheral -- we rethink our web service interface in terms of URIs, HTTP, and XML. It is this rethinking that takes our web services beyond the capabilities of the first generation technologies based on Remote Procedure Call APIs like SOAP-RPC.

In this article I discuss the applicability to REST of several industry buzzwords such as reliability, orchestration, security, asynchrony, and auditing. Intuitively, it seems that the Web technologies are not sophisticated enough to handle the requirements for large-scale inter-business commerce. Those who think of HTTP as a simple, unidirectional GET and POST protocol will be especially surprised to learn how sophisticated it can be.

Quick Review

REST is a model for distributed computing. It is the one used by the world's biggest distributed computing application, the Web. When applied to web services technologies, it usually depends on a trio of technologies designed to be extremely extensible: XML, URIs, and HTTP. XML's extensibility should be obvious to most, but the other two may not be.

URIs are also extensible: there are an infinite number of possible URIs. More importantly, they can apply to an infinite number of logical entities called "resources." URIs are just the names and addresses of resources. Some REST advocates call the process of bringing your applications into this model "resource modeling." This process is not yet as formal as object oriented modeling or entity-relation modeling, but it is related.

The strength and flexibility of REST comes from the pervasive use of URIs. This point cannot be over-emphasized. When the Web was invented it had three components: HTML, which was about the worst markup language of its day (other than being simple); HTTP, which was the most primitive protocol of its day (other than being simple), and URIs (then called URLs), which were the only generalized, universal naming and addressing mechanism in use on the Internet. Why did the Web succeed? Not because of HTML and not because of HTTP. Those standards were merely shells for URIs.

HTTP's extensibility stems primarily from the ability to distribute any payload with headers, using predefined or (occasionally) new methods. What makes HTTP really special among all protocols, however, is its built-in support for URIs and resources. URIs are the defining characteristic of the Web: the mojo that makes it work and scale. HTTP as a protocol keeps them front and center by defining all methods as operations on URI-addressed resources.

Auditing and Securing REST Web Services

The most decisive difference between web services and previous distributed computing problems is that web services must be designed to work across organizational boundaries. Of course, this is also one of the defining characteristics of the Web. This constraint has serious implications with respect to security, auditing, and performance.

Does REST make sense for you? Share your views in our forum.

Post your comments

REST first benefits security in a sort of sociological manner. Where RPC protocols try as hard as possible to make the network look as if it is not there, REST requires you to design a network interface in terms of URIs and resources (increasingly XML resources). REST says: "network programming is different than desktop programming -- deal with it!" Whereas RPC interfaces encourage you to view incoming messages as method parameters to be passed directly and automatically to programs, REST requires a certain disconnect between the interface (which is REST-oriented) and the implementation (which is usually object-oriented).

In the previous article I discussed how it is possible to use Access Control Lists (ACLs) to secure services which use URIs as their organizational model. It is much harder to secure an RPC-based system where the addressing model is proprietary and expressed in arbitrary parameters, rather than being group together in a single URI.

With URI-based (REST) web services, administrators can apply ACLs to the service itself and to every document that passes through the service, because each of them would have a URI. As two business partners work their way through a process, each step could be represented by a new document with an attached ACL. Other partners (or auditors) could be given access to this document later merely by manipulating the ACLs.

In the REST model, both business partners would have a shared view of the URI-space representing the process. Rather than sending each other business documents through an email-style pipe, they would PUT them on each other's web sites with shared URIs and passwords. These could be easily checked for discrepancies. Third parties can be brought into the process (perhaps for auditing) merely by pointing them at one or both of the URIs. Standard HTTPS and HTTP authentication and authorization would be sufficient to keep intruders from also being able to look at the documents.

Of course, HTTP-based Web Services can go through firewalls easily, but that is the only point of similarity with RPC tunneling through HTTP such as XML-RPC or SOAP-RPC over HTTP. When you use HTTP over a firewall, you are being very explicit about what is going on. Your system administrator can look at the logs to determine what services are running and who is accessing them. She can disable PUT or POST to make certain parts of the service read-only. She can use standard filtering and hacker detection tools. When you use HTTP *as* HTTP, you and the system administrator are on the same team.

Conversely, when you tunnel XML-RPC or SOAP on top of HTTP, just to get through a firewall, you are deliberately subverting her work and reducing the efficacy of her tools. Because you are hiding your real actions in the XML body, you make it much harder for her to use standard filtering tools. This greatly expands the opportunity for new security holes. Famed security expert Bruce Schneier has spoken out against SOAP over HTTP for exactly this reason.

Service Context

People building web services often complain that the "tricky bit" of web services is maintaining

shared context. For instance, context might include the following.

- What organization does the client program represent?
- Where are we in this business process?
- What transactions have we done in the past?
- Are there any resources that I promise to hold for you?
- Are there any notifications I promise to deliver to you later?
- What permissions do you have?

There are three main ways that two partners can share context. One is to send the entire context with every message. This is obviously not very scalable: as the relationship deepens the context will grow larger and larger. Another option is to merely require each partner to keep context privately and presume that the other partner has the same idea of context. As you can imagine, this is quite unreliable: a network hiccup or programming bug could make the contexts diverge. The mechanism used on the Web today is to assign URIs to the context. For instance, on Expedia there is a "My Itinerary" URI for each individual. Within that, every purchase you have recently made has its own URI. While you are purchasing a new ticket, each step in the process is represented by another URI. The client may keep copies of resources for its own protection, but the context is always mutually available as a series of linked documents at a URI.

Orchestration

Every method that can be invoked on a resource or service is a possible connector between the client and the service. If every service has a hundred *different* methods then your connections become very complex -- essentially point-to-point integrations rather than reusable patterns.

There are various systems in the computing world that have proven the power of having just a few methods rather than many. For instance, every true Unix hacker knows that the command line is incredibly powerful because it is possible to pipe data from one process to another using the redirection "methods", ">", ">>", "<". The other Unix command line tools act as standardized filters and transformers connected by these methods.

Similarly, if you think of a SQL table as a resource, the methods SQL makes available are only SELECT, UPDATE, INSERT and DELETE. The rest of SQL is a set of transformers and filters that allow you to combine these methods into services. .NET My Services has Query, Insert, Replace, Update and Delete. As I showed in the last article, UDDI has get_*, delete_* and save_*. This pattern is ubiquitous: a small number of methods applied to diverse kinds of data.

HTTP has GET, PUT, POST, and DELETE. Anything that can be done with SOAP RPC or any other RPC can be done with those four methods. In fact, it is precisely because HTTP has few methods that HTTP clients and servers can grow and be extended independently without confusing each other. Rather than invent new methods they find ways to represent new concepts in data structures (increasingly XML data structures) and headers.

Now that we've boiled down our system to these basic methods, it turns out that we have the beginnings of a web service coordination, orchestration, and assembly language. I could imagine defining a new web service as easily as:

```
i = GET http://www.stockquotes.com/xmlquotes?method=IBM
m = GET http://www.stockquotes.com/xmlquotes?method=MSFT
if i > m:
    WITH AUTHENTICATION $myuserid $mypassword
    POST http://www.etrade.com/stocks/IBM
else:
    WITH AUTHENTICATION $myuserid $mypassword
    POST http://www.etrade.com/stocks/MSFT
```

Or maybe we don't need a new language: we could incorporate these principles into existing scripting languages. The point is that unifying the method vocabulary of the Web provides tremendous opportunities for simplifying interactions. Nobody learning a new web service would ever have to learn the semantics of the various methods again. Web services can be combined through simple Unix-style pipes: GET this, GET that, transform, PUT there.

Of course there is no free lunch. Using someone else's web service requires you to understand their data structures (XML vocabulary and links between documents). But this is true whether we use REST or SOAP RPC. RPC APIs merely hide the problem behind an extra layer of non-standardization. First you must figure out the method names available. Then you must still figure out the data structures that may be used as parameters. And then behind those data structures is the implicit data model of the web service.

UDDI has an implicit relational data model. .NET My Services has a concept of a "virtual document". The Web already has a data model of URI-addressed resources connected by hyperlinks. This shared data model has already been implemented in thousands of products, and already allows for startling levels of coordination. Consider a BabelFish translation of a Google result page aggregating multiple otherwise unrelated resources.

URI-centric web services are inherently easier to integrate because the output from one service can be easily used as the input to another. You do this merely by supplying its URI. There is nothing magical or special about this. It is how programs on your desktop computer share information today. The amazing thing is that specifications like SOAP and WSDL have no real support for this simple but powerful data sharing mechanism.

Once you begin to orchestrate multiple web services, transaction processing becomes much harder. It is very difficult for the client to get the various services to have a common view of a "transaction" so that a failure on any service causes a complete rollback on all of them. HTTP does not have a magical solution to this problem but neither do specifications such as SOAP or ebXML. The solutions proposed by the OASIS Business Transactions working group are currently protocol agnostic and should work fine with HTTP.

Asynchrony

People often complain that HTTP is not asynchronous. Unfortunately they often mean different things by that term. Most often they compare it to SMTP as an example of an asynchronous protocol. However, you could make the case that both HTTP and SMTP are synchronous or asynchronous depending on how you define those terms.

Email is asynchronous not primarily because of SMTP itself, but because of the collection of software that does store-and-forward, failure notification and replies. This software can be reused in an HTTP-services world through gateways. HTTP->mail gateways will be necessary for these packages in order to support SOAP-over-HTTP to SOAP-over-SMTP gateways regardless.

Still, HTTP does not have perfect support for asynchrony. It primarily lacks a concept of "callback" (also called a "notification" or "reply address"). Although this has been done many times in many places, there is no single standardized way to do this. Software dealing with notifications is not as reusable as other HTTP-based software modules. There is work under progress to correct this situation, under the name

HTTPEvents.

HTTP needs concepts of explicit and implicit store and forward relays, transformational intermediaries, and return paths. These may be borrowed from SOAP headers and SOAP routing. In fact, some REST proponents believe that this is the only part of SOAP that is strongly compatible with the Web and HTTP.

Many commercial applications have been built using HTTP in a peer-to-peer, asynchronous fashion, by companies as large as Microsoft to as small as KnowNow. Now there is active effort standardizing best practices.

Reliability

Networks are inherently unreliable and the Internet is an extreme case of this. You must build network-based software to cope with failure, no matter what protocol you are using. Nevertheless, the right combination of software and protocols can make message delivery more reliable than it would otherwise be. What most people ask for is that a message be delivered if at all possible, and be delivered at most once. If it is not possible to deliver it, then it should be reported to the application.

Writing reliable-delivery software with HTTP is relatively easy. Although it is not rocket science, a full description does take more than a couple of paragraphs.

The bottom line is that software written on top of HTTP can make all of the same guarantees that expensive message queuing software can -- but on top of a protocol that you can send across the Internet to standards-compliant business partners. HTTP applications have traditionally not gone to this level of engineering effort but the underlying protocol does not prevent it. If you are not bothered by the fact that most reliable messaging software uses proprietary protocols then it is easy to tunnel HTTP on top of that proprietary protocol for use within a single organization.

Understanding Addressing

Hopefully I have demonstrated that REST can address all of the industry buzzwords successfully. Now let me ask for your help in promoting *addressing* to the same level. We as an industry need to understand that this is not just *an* issue, it is *the* issue. Until we get it right, we cannot expect to make progress on other issues. As I have shown, it is the basis of security, web service orchestration, and combination. It is the difference between a unified web services Web and a balkanized one.

For instance, a balkanized way to reference about a particular stock value is as `WebService.stockQuote("KREM")`. This syntax is particular to some programming language and is not available outside of it. It can only be used by some other service through some form of glue. A universally addressable way is "http://www..../webservice/stockquotes?KREM". This can be accessed by any application anywhere in the world (with proper security credentials) whether or not it was written to understand stock quotes.

A balkanized way of submitting a purchase order is to call an RPC end-point which returns a corporation-specific purchase order identifier (even a UUID). A universally addressable way is to ask the server to generate a new location (using POST) where the purchase order can be uploaded (using PUT). Thereafter it can be referenced in legal materials, annotated with RDF or XLink, secured with ACLs and used as the input to other services. Once the purchase order is given a URI it becomes part of a World Wide Web of documents and services.

Since the dawn of the Web, proprietary addressing schemes have been swept away or rendered irrelevant in the face of the URI onslaught. Examples include Hyper-G, Archie and Microsoft Blackbird. Competing with URIs is neither healthy nor productive. Other addressing schemes survive on closed networks with a sort of second class status: AOL keywords, Windows "Universal Naming Convention" (UNC) names, "Windows Internet Naming" and so forth.

Now consider that every single RPC service by definition sets up its own addressing scheme and data model. If history repeats itself we can expect each RPC-based service to be relegated to second class syntax in comparison to their competitors that embrace the universal addressing scheme provided by URIs. More likely, all services will provide both interfaces (as, for example, better UDDI implementations do) and the RPC interface will just fall into disuse. "Full of Sound and Fury. Signifying Nothing."

Case Studies

Despite its advantages, HTTP-based, URI-centric resource modeling is not a common way of thinking about networking issues and REST-based web services (other than web *sites*) are not very common. On the other hand, useful, scalable, public RPC-based web services are also quite difficult to find.

The most obvious examples of HTTP-based web services are regular web sites. Any site that presents a purchasing process as a series of web pages can trivially be changed to do the same thing with XML. People who go through this process get all of the benefits of REST web services and none of the expense of re-implementing their business logic around a SOAP-RPC model.

Two businesses that have created (admittedly simple) REST web services are Google and O'Reilly. Google offers to its paid subscribers the ability to have search results published as XML rather than HTML. This makes it easy to build various sorts of sophisticated programs on top of Google without worrying about shifting HTML formats.

The Meerkat Example

O'Reilly's Meerkat is one of a very few useful, public web services. Unlike the majority of the services described on XMethods, Meerkat is used by thousands of sites every single day.

Meerkat uses the three foundation technologies of second-generation web services. It uses a standardized XML vocabulary: RSS. Meerkat would never have become as powerful and scalable if it had invented its own vocabulary. It absolutely depends on the fact that it can integrate information from hundreds of sites that use the RSS vocabulary and the HTTP protocol.

In addition to using HTTP and RSS, Meerkat uses URIs as its addressing scheme. It has a very sophisticated URI-based "API".

Meerkat's content is also available through an XML-RPC API. Before the REST philosophy was popularized it was not clear that Meerkat's HTTP/XML-based interface was already a complete web service. It would be an interesting project to compare and contrast these two interfaces formally.

One interesting point, however, is that all of Meerkat's content aggregation is done through HTTP, not XML-RPC or SOAP. It would be ludicrous to suggest that every content publisher in the world should not only support XML-RPC and SOAP but also some particular set of methods. This would be the situation if instead of inventing the RSS vocabulary the world had standardized the "RSS API."

To be fair, HTTP's advantages would have been less pronounced if Meerkat's interaction with these sites had required two-way communication instead of a simple one-way information fetch. At that point you do need some glue code or at least an orchestration language.

Meerkat shows that when many sites share an XML vocabulary, a protocol and a URI namespace, new services can arise organically. It is arguably the first equivalent in the Web Services world to a large-scale, distributed service like Yahoo. Meerkat's success suggests strongly that the most important enabler of large-scale, distributed web services will be common XML vocabularies.

REST limitations

There is no free lunch. REST is not a panacea. The biggest problem most will have with REST is that it requires you to rethink your problem in terms of manipulations of addressable resources instead of method calls to a component. Of course you may actually implement it on the server side however you want. But the API you communicate to your clients should be in terms of HTTP manipulations on XML documents addressed by URIs, not in terms of method calls with parameters.

Your customers may well prefer a component-based interface to a REST interface. Programmers are more used to APIs and APIs are better integrated into existing programming languages. For client-side programmers, REST is somewhat of a departure although for server-side programmers it is not much different than what they have been doing for the last several years, building web sites.

REST is about imposing a programming discipline of many URIs and few methods. RPC allows you to structure your application however it feels best to you. Let it all hang out! If a particular problem can be solved with RPC, and future extensibility and security are not going to be issues for you, you should certainly use the looser approach.

HTTP is also not appropriate in some circumstances. Because HTTP runs on top of TCP, it can have high connection times compared to protocols intended first and foremost for efficiency. HTTP is designed primarily for the kind of coarse-grained interactions that are used on the public internet, not the kind of fine-grained ones that might be appropriate on a single desktop, within a department or even in certain enterprise situations.

Once again, if DCOM or CORBA solves your fine-grained problem then there is no reason to move to REST. In my opinion, REST will first dominate primarily in the world of partner-facing, external Web Services. Once this happens, it will start to migrate to the Intranet, just as the Web did.

The Best Part

The best part about REST is that it frees you from waiting for standards like SOAP and WSDL to mature. You do not need them. You can do REST today, using W3C and IETF standards that range in age from 10 years (URIs) to 3 years (HTTP 1.1).

Whether you start working on partner-facing web services now or in two years, the difficult part will be aligning your business documents and business processes with your partners'. The technology you use to move bits from place to place is not important. The business-specific document and process modeling is.

There is no doubt that we need more standards to make partner-facing web services into a commodity instead of an engineering project. But what we need are electronic business standards, not more RPC plumbing. Expect the relevant standards not to come out of mammoth software vendors, but out of industrial consortia staffed by people who understand your industry and your business problems -- and not networking protocol wonks like Don Box and myself.

REST does not offer a magic bullet for business process integration either. What REST brings to the table is merely freedom from the tyranny of proprietary addressing models buried in RPC parameters. Do not be afraid to use hyperlinks and URI addresses in your business documents and processes. Specifications like SOAP and WSDL may make that near impossible, but that is a problem with those specifications, not with your understanding of your problem. If you use hyperlinks in your business document and process modeling (as you should) then there is a protocol that does not get in your way: HTTP.

REST: Everything Old is New Again

The rhetoric around web services describes them as "like the web, but for machine to machine communications." They are said to be a mechanism for publishing processes as the Web published data. REST turns the rhetoric into reality. With REST you really do think of web services as a means of publishing information, components and processes. And you really do use the technologies and architecture that make the Web so effective, in particular URIs, HTTP and now XML.

Thanks to Jeff Bone and Mark Baker for some core ideas.