DOCUMENT TITLE: REST Oriented Architectures (ROA): Taking a Resourceful Approach to Web Data
DOC TYPE: <Unsure>
THREAD: Data Management Strategies
ANALYST(S): Kurt Cagle
ADDITIONAL INPUT: Peter O'KellyVERSION #: 1
KEYWORDS: resource, REST, XML, Data, Data Modeling, ROA, SOA, Syndication, XQuery, RIA, XForms
TOPIC MAPS: DMS: Data management, DMS: Database management, DMS: Technologies and Standards
DOCALERT TEXT: Resource Oriented Architectures represent a significant shift in web application development, creating a resource-centric rather than process-centric view of the web that aligns more closely with the way the Internet itself is designed.

Synopsis

Resource Oriented Architectures use the fundamental characteristics of the web itself in order to provide and update content on the web. While much of the philosophy concerning REST has been around since the early 1990s, the tools for turning these philosophies into working systems are only now becoming feasible. The fundamental tenets of ROA – that the web itself is primarily a giant database, that resources are abstractions that can be manifest in different representations, that a query-oriented resource architecture is more robust than a verb-oriented services architecture, and that a common publishing and syndication protocol is necessary to make such an architectural system work – are being adopted increasingly by people who realize that services oriented architecture are not effective at

getting data from users or providing it to them in an easy to use way, but that ROA can do precisely that.

Such a shift in perception is necessary but will nonetheless take a while to happen. It's necessary because the amount of information on the web is piling up faster than it can be indexed, and because under the current architectures the cost of developing "editors" for that data is prohibitive compared to the value of that information. It's necessary because the data within organizations is getting more complex than can be readily handled with a name/value approach to application development, and is increasingly contained within non-traditional data sources – Excel spreadsheets or Microsoft Word documents, for instance, or external data streams.

Adoption will take time, however, because such an approach reduces the competitive barrier impedence that corporations can utilize to sell services, because it will take time to educate people in the underlying technologies and because there is a long-standing belief that ROA and SOA systems are incompatible. The rapidity at which companies lined up behind AtomPub, on the other hand, points to the fact that many IT organizations recognize the value to themselves that an AtomPub-type architecture opens up, while the educational curve is frankly true of most technologies – it will happen, slower than its proponents may hope but faster than its critics anticipate.

Analysis

Do you know where your data will be tomorrow? Not too long ago, such a question would have been easy to answer – your data would be locked away, safe and secure in a relational database, accessible by specialized applications that are written by programmers working closely with your database administrator. Chances are good that this may still be the case today, though no doubt the people within your organization are beginning to clamor for more unfettered access and are asking for more applications that can let them do analytics on the data, to present that data to the outside world in some form of web service, or to tie that data in with other data repositories.

Yet, the question of data location is not a trivial one. How much of the information in your domain is in the form of annual reports or cost/benefit analyses? How many sales projections are to be found in Excel spreadsheets? For that matter, how much of your customer contact information can be found in a mySQL database in the web department that's otherwise inaccessible to your particular database (and similarly, how much of the internal sales numbers that you would like to provide for a web presentation are locked up in an Oracle database in the basement that's otherwise inaccessible to you as a developer … or is located in your data warehouse in Tulsa, Oklahoma). How much of the information comes from external data repositories on the web?

In the Cloud: ROA and the Internet Information Model

The last decade has seen a fairly radical change in the concept of what both information and data are, with most of that change being driven by the use of data systems to drive the production of web content, and increasingly for the web to act as a means to create, modify and manipulate data to and from data stores. This latter shift has accompanied a similar change in thinking about the web less as a way to get web pages and more to exchange the serializations of data structures as messages across the Internet. Indeed, this latter activity, in one form, lies at the heart of what are known as services oriented architectures, or SOAs.

The use of data "messages" is certainly not new – messaging languages such as the various Electronic Data Interchange formats (EDI) have been around since the early 1970s, and messaging is considered an integral part of most contemporary enterprise service bus stacks. However, in most cases the approach that has been taken with contemporary message formats and systems (such as SOAP / WSDL based systems) has in general been to treat the modern day analog to EDI ports, the web server, as being an object with associated methods designed to either treat the messages as object method invocations (usually to be marshaled into binary objects), with the result of these messages in turn sent back to the sender to be serialized as an object on the client.

Recently, however, software and database developers have begun to question whether, in the course of attempting to turn the web into a large, complex application, they've somehow missed out on a very important (and subtle) idea: *What if the web is actually just a giant database?*

This idea has gained currency as people developing rich Internet applications (RIAs) using AJAX techniques have begun to question the notion that the Web's only use is to serve up HTML and media files. Grab a collection of upcoming events and serialize them in an iCal format, and you have the basis for building an event calendar, where the client does the graphical

layout and uses iCal (or its XML analog xCal) as its data source. Ask for a collection of news items from a news feed URL and serialize that as RSS or Atom and you can use them to create a just-in-time newspaper. Stream it as geospatial information in Keyhole Markup (KML) and you can build an application in GoogleEarth.

Each one of those examples involve making a request from a web URL, a request that will come back as serialized data. In some cases, that web URL may be an explicit query or method invocation (at least at that level of abstraction, there's no real difference between the two), while in others the URL holds static information or information that may change but does so independent of any parameters. There's typically no SQL involved here at the interface level, no inner or outer joins, no references to tables. Yet at a very deep level, the Web is fundamentally a place to retrieve data, and increasingly is the place to store it.

This realization, of the web as database, is shaking the foundations of distributed application development. It is bringing one of the more enigmatic aspects of database development – data abstraction – into sharp relief not only as a handy tool for programmers but rather a necessary, perhaps even vital design component for nearly all web applications.

This realization in turn is driving awareness of new technologies that need to exist in an "application stack" for the web to be used in such a database fashion, particular to deal with the web as a database of *resources*. This application stack works on the principle that by working with resources through the use of typical database-like atomic transactions – getting resources from a location, putting resources back to that location, adding a resource to a collection of resources, deleting resources and so forth – that you can build applications that are considerably more robust, stable, scalable and easier to program with.

The general model so proposed is what is becoming known as Resource Oriented Architectures (or **ROA**s), a term first proposed by Sam Ruby and Leonard Richardson in Restful Web Services (O'Reilly, 2007). The specific application stack covered in this paper (and the use of Atom and AtomPub as a primary component for publication) was also first proposed by Ruby, but has been expanded considerably in this paper.

The ROA Stack as discussed here covers the following principles:

- The Internet Information Model, which treats the web as collections of resources bound with data publishing metadata (REST).
- The deployment of a standardized "publishing" mechanism that performs atomic transactions (GET, PUT, POST, DELETE, HEAD) on resources and resource collections. (e.g., AtomPub)
- The serialization of resources and resource collections as metadata bundles wrapping either content or links to representations of content (e.g., Atom)
- The elaboration of a linking mechanism for both retrieving and modifying resources and to establish additional relationships between resources. (e.g., XLink)
- The introduction of categorizations into resource metadata to provide multiple organizing principles of resources themselves. (e.g., Domain specific/Atom)
- The use of syndication principles to provide asynchronous notifications across a distributed environment. (e.g., Atom/AtomPub)
- The introduction of an abstract data query environment that can work to integrate and aggregate relational information, structured and semi-structure, streaming content and computed data in order to filter resources and potentially shape output content. (e.g., XQuery/XSLT2)
- The use of stream-capable rich Internet application clients to act as both data consumers and data editors in either component or model/view/controller orientations. (e.g., XForms)

This paper recognizes that there may be multiple potential implementations of an ROA stack. The focus covered here is to look primarily at those

technologies that correspond (more or less) with the W3C application stack. A future paper may explore in more detail similar stacks built around the Java or .NET environments.

Defining Resources

Most people have an intuitive notion of what a resource is – it's a thing, an object, an entity – but within the context of the Web resources have a considerably more specialized meaning. Just as a record in a database has a considerably different meaning to a SQL developer than it does elsewhere, understanding the meaning of resources and collections of resources, the Internet Information Model or IIM, is key to understanding how data itself can increasingly come from "The Cloud" of the Web.

Consider a book, say Snow Crash by Neil Stephenson, a rather dark and disturbing look into a world that's increasingly looking like our own. What does it mean to put that book on the Web? Well, it could in fact mean any (or all, or none) of the following:

- Someone gets the manuscript from the publisher and makes that available as a HTML file at a certain URL.
- Someone runs the book through a scanner and converts each page into a page in a PDF.
- You click a link that takes you to Amazon.com, where you pay for the book and the book then appears in your mailbox in three business days.
- Stephenson reasons the novel as a sound file and puts that up on the web for download as streaming audio.
- The Snow Crash players do a video presentation based upon the book and put it up on YouTube.
- From a library collection site with Snow Crash in its catalog, you put a hold on the item in order to pick it up when it becomes available.

Each of these things can be said to be a *representation* of the book. Interestingly, this means that even the printed copy of the book from Amazon is a representation of the book via the web, albeit one working across the transport protocol called Federal Express.

(The term REST, used frequently in this paper, is a shorthand notation for Representational State Transfer, and in general is meant to indicate the use of a resource model oriented system for both retrieving content and (in the case of AtomPub) updating that content..)

What this implies is that a resource is an abstraction of a particular object, one that can have multiple potential representations while still being the "same" object. This is a notion that is familiar to data modelers and librarians, who both recognize that there's a distinction between a particular book being in the system and having a copy of that book in the system. Each copy is undeniably unique (it's a physical object that will either be in one branch of a branch library, borrowed, in transit, or possibly lost or destroyed) yet each of the copies is still "Snow Crash" and can be substituted for a different copy with no loss of fidelity to the user.

Yet this points out as well that both the resource and each particular copy of that resource have unique identifiers which identify them in the system; when a patron requests "Snow Crash" he is only looking for the copy that is most convenient to at patron at the time, and otherwise couldn't care less whether or not it is copy #1 or copy #92. In the case of the HTML file, the PDF and the streaming audio, the id of that resource is likely a Uniform Resource Locator (or URL), which, as the name implies, is a locator (an address) for the resource on the web.

What you get back when you request the resource from a given address is actually a two part message. The first part is the header, and it generally

does not get displayed within a web browser. The header consists of a basic set of "publishing metadata":

A title (a filename)
When was the resource first published to the web
When was the resource most recently updated
The local path to the representation on the server
The file size of the resource in question
The account name of the agent who most recently updated the resource
The mime-type or content-type of the resource
Additional attributes, as defined by the server

The body, in turn, contains the actual bytes of data that, when passed to the appropriate user-agent, will display the content as a web page, a graphic, a playlist or any other entity. This body is otherwise known as the *representation* of that resource.

One key point in the understanding of resources within the Internet Information Model is that the specific content of the resource is secondary to the existence of this metadata record for that resource. This is worth restating:

> ***In the emerging Internet Information Model, all resources have at least a block of common metadata that describes their "publishing" characteristics, including categorizations, that has the same structure – regardless of what that resource is. By leveraging this common metadata, applications can work with this information as a proxy for the resource itself.***

Typically, within a database query, the resulting record-set frequently contains similarly metadata that indicates the currency and agency of the information involved, though it usually needs to be tracked explicitly rather than being an implicit part of the web.

One of the most important pieces of this information is a key that can identify that particular record in subsequent queries. With such a key, the information in that record is said to be *addressable*, at least within the context of the database – given the address, you should be able to retrieve the record.

A web resource is, by definition, addressable, through the mechanism of the Uniform Resource Locator or **URL**. Since the Internet as database is, in a very real sense, global, this means that such URLs can be used to identify resources anywhere on the web.

One consequence of this notion of resource URL is that if a database is given a URL and a given conceptual record within that database has a corresponding key that's "local" to the scope of the database, then you could use both URL and key to uniquely identify a specific record as a web resource.

This has some profound implications. If you have a serialization "bridge" that can convert such a record into a more readily transportable representation (such as XML or JSON) then this means that you can "publish" your data to the web in the same way that any other resource is published, and such a record can make use of the underlying resource bias of the web.

Building Relationships with Links

An address by itself is of significance primarily due to a second key structure of the Internet Information Model – the link. Most people have an intrinsic understanding of links from their use within web pages, where such links are usually associated with click such that "clicking on a link" (actually a marker for the link) will cause the linked page to be loaded into the browser.

However, these are also the simplest forms that links can take, and people tend to assume that the role of a link is to retrieve a representation. In point of fact links can be considerably more sophisticated. First, a link never exists by itself – it is in fact a relationship between a containing resource and the linked resource (hence implying that a resource may "hold" one or more such links). A link typically combines at least four facets:

An address where a given resource is located.
A relationship that describes the significance of the link to the containing resource.
A modality that indicates the expected action upon "activation" of the link.
An abstract bundle that provides some human level relevance to the link itself.

While it is possible to have fairly esoteric links (link collections, bi-directional links, and so forth) in a distributed environment it is usually better to replace these with the appropriate complement of simple, one-way links.

In Web 1.0, the only link that a given resource could have in its header is a reflexive one identifying the location of the resource itself. More typically, it was the representation of the resource (its content) that contained such links, with the specific implementation of such links being dependent upon the nature of the resource itself.

Links serve to make resource networks act like data models, and not all links serve as a signal to retrieve content. A book may contain a link to a specific library (in a one to one relationship) in a relationship that indicates that the book "belongs to" that library, while a separate link cold be made to a patron where the book "is borrowed by" the patron. Note that in this case all three of these things – book, library and patron - are themselves resources.

One of the most significant changes in recent years has been the ability to associate such links to a resource as part of the metadata about the resource rather than as relying upon the representation to hold these things. By making these metadata properties rather than data properties, more complex data models can be established for sets of resources without explicitly needing to parse the resource representation, which can be especially useful when the representation is a media file such as an image or sound resource.

A particularly intriguing consequence of such meta-links is that it opens up the idea that a resource can be an abstract entity with multiple representations; such a resource becomes the analog to a library catalog card – it is a pure bundle of metadata with links to each different representation of the book in question. The interesting thing with such an "abstract" resource is that it can itself be serialized – as XML, as JSON, as other such formats. Indeed, this concept, that the metadata record of a resource can itself be serialized as a representation is what makes resource oriented architectures (ROAs) feasible.

Dealing with Collections

Most people's experiences when working with resources on the web, unlike those in databases, is that they deal with them one at a time. You go to a web page, you download an application, you view a single image or media file. This is one of the main reasons why it is sometimes difficult to understand the importance of collections on the web.

Another impediment towards this understanding comes from people's familiarity with the folder/file metaphor that is so pervasive on the desktop (and indeed is embedded deep within even the command line). A collection is a folder of either files or other folders, and you can thus navigate through the contents of a file system by following the folder links. This provides the

implication that navigation on the web mirrors this structure – with the power of a named collection in particular being that it defines linked resources as “positional” siblings.

In a database, on the other hand, what is commonly thought of as a collection is more analogous to the results of a record set generated from a SQL query. While it is possible to apply the file/folder metaphor, most applications treat such a collection simply as a (potentially infinite) sequence of items related by the query. Indeed, a SQL view, in which you associate a given SQL query with a name, encapsulates the concept of collections nicely.

Significantly, the Internet information model actually works well with both of these interpretations of collections. Consider, for instance a scenario where you have a “collection” of books in an online library. The specific representation of the books at this stage is unimportant, suffice it only to say that such a representation exists. This collection could be thought of as being contained within a “folder” representing the library itself.

However, in the resource model, such containment is largely illusionary. From the standpoint of the web, all resources exist as part of a set, and there’s no explicit hierarchy that is imposed upon that set. Perhaps a better way of thinking about this containment model is to imagine that one of the resources that exists is a list of all of the books in the library, with links to their respective resources (it may very well be a book itself, mind you, but down that slope lies madness). Another resource that exists may be the list of all books in the Science Fiction genre. This obviously is contained in the first set, so the containment model looks good. Suppose, however that a third resource was a list of all books written by authors whose name starts with “S”.

Granted, as an organizing principle, such a library collection might seem a bit random, but the point here is that both lists are potentially valid, even

though they both contain Neil Stephenson's Snow Crash. Collections are not exclusive. Rather, *collections on the web are arbitrary groupings of resource links (likely with some clarifying metadata) that are themselves resources.*

One implication of this is profoundly important in understanding restful architectures: because the representation of any resource may in fact be generated from some external process, this means that a collection can be generated in one or more ways:

in response to internally changing state variables (e.g., time),
aggregation due to categorizations,
or through the agency of a query operation.

The specific implementation of this automation is unimportant – this particular detail gets lost because the URL in question serves to abstract out that automation. This means that such resource based collections can work regardless of what language is used for serving the content, which in turn implies that resource oriented architectures should (in theory) be implementation independent. In point of fact there are usually some basic processing capabilities that the server needs in order to generate this content – a purely static set of collections will not give you the ability to work with dynamic content – but beyond that obvious limitation this property of resource systems make them attractive in heterogeneous environments.

It's worth digging a little deeper into each particular generator, as they describe in very broad detail whole classes of RESTful applications.

Collections as Feeds

The web is dynamic. Content is added to it, is modified, is removed. This phenomenon is most clearly seen with newsfeeds. A newsfeed consists of a collection of related web pages (where each page is a resource) listed in reverse time order of publication. This particular ordering makes a great

deal of sense, as in general the most recent publications are most likely to be the ones of immediate interest when reading the news (which is why it's called "new-s", of course). Such collections are also frequently called syndication feeds, and they will likely play a major role moving forward as the resources associated with them move beyond web page content and into more record-oriented content.

Collections determined by Taxonomies

A *taxonomy* can be thought of as a set of terms (a vocabulary), coupled with links that describe the relationships between the terms. Taxonomies can be found that fall into a number of different structures:

Unordered lists. This is simply a bag of terms, with no particular ordering implicit in them. Tagging of web content is an example of this particular type of taxonomy.

Ordered lists. This particular taxonomy creates terms that act as transitional state categories. For instance, a workflow from "draft" to "reviewed" to "published" to "archived" represent different states in an ordered list.

Hierarchies. For many people, taxonomies are synonymous with hierarchies, largely due to familiarization with biological (Linnaeus) taxonomies. A hierarchical taxonomy basically creates sets of buckets (or folders) that a given resource can be contained in. Directory systems provide one kind of hierarchical taxonomy (if you limit a resource to having only one term in that particular taxonomy), but even if a given resource has more than one such term this provides the foundations for a powerful tree traversal navigational system.

Networked. A hierarchy is a specialized form of a networked taxonomy, where each term is connected to other terms with a more general relational model than an "is child of" or "is parent of" relationship. A general networked taxonomy often works well when the vocabulary is open-ended and the relationship consequently are assigned by the community using the taxonomy.

The concept of introducing categorization into the metadata of a resource record is relatively new – this was not in fact a part of the original

specification for HTTP 1.0. In general, such a categorization contains three parts:

Scheme. The schema identifies which taxonomy is to be used, either by providing a namespace or by indicating a specific organization. Given the large number of taxonomic schemes in place, use of the scheme identifier is necessary.

Term. A term in the scheme is the specific work or set of words that identifies membership in the scheme. Note that a term is essentially a token, and does not necessarily have to be a recognizable word – it only needs to exist as a vocabulary term within the schema.

Label. The label provides a human readable equivalent for the term if its not necessarily decipherable on its own.

In the emerging Internet Information Model, a given resource can have zero or more categories, in general with no constraints on whether or not two or more terms can be in the same scheme. External modeling considerations may have an influence there, of course, but these are outside of the scope of the resource model itself.

Categorizations will likely play a huge roll in RESTful systems, both because they separate the question of topical organization from physical storage and because categorizations make it possible for resources to be organized along several “axes” simultaneously. For instance, for a set of movie resources, one taxonomy could establish the rating of the movie (G, PG, PG-13, R or X), another could establish the genre of the movie, a third could organize it by country of origin and so forth. By moving this information into categorizations rather than storing it in the data for the representation of the resource, such organization can be accomplished with a trivial amount of work.

Collections through Queries

Creating a resource collection via queries is very similar to creating a record-set using SQL – you use some form of query abstraction language in order to

generate a collection of resource pointers that satisfy a given set of constraints. The specific serialization of that collection is largely a matter of preference, though both XML serializations (such as the Atom specification, covered later or SOAP Response messages) and JavaScript Object Notation (JSON) have become largely the de facto serialization mechanisms for use on the web, although HTML serializations are not unknown (such as used by XOXO or similar bookmarking schemas).

The precise nature of the query abstraction language requires a bit of thought, however. For all that there are similarities between the SQL relational model and the resource-centric IIM, there are also some significant differences. A SQL query will result, in general, in a recordset consisting of records having a linear sequence of name/value pairs, usually formed by performing logical joins between relational tables on PK/FK intersections.

Resource representations on the other hand, have no guarantee that they will be given in the same format – and indeed, such resources will usually either be some HTML/XML documents or will be a binary representation of media or application content, though of course, text files, source code files and so forth are also not unheard of.

This means that queries will take place at one of two levels – the query will either be done on some aspect of the metadata bundle (such as categorizations, publication information, or indexed searches on summary content) or will take place on the structured representations in those cases where the representations can effectively be queried.

XQuery and Data in Motion

One of the likely contenders for such a query language is the recently published (Feb. 2007) W3C XQuery specification. This language is optimally designed for working with XML, but the underlying data algebra for the models are, in many cases, simply an elaboration upon the data algebra for

SQL nearly thirty years before, taking into account both an awareness that data is moving out of the database, becoming manifest as streams of data rather than stored data, and that an introduction of namespaces (similar to schemes on the categorization side) is necessary to create more of a modularization of query capabilities and to handle vendor customization in a standards-friendly manner.

For all that, XQuery as a language can be used with other data formats so long as these formats can be mapped into XML-like structures. This means that XQuery can be tied into SQL databases (as has been done with most of the major SQL RDBMS vendors) and can similarly generate output content that can be serialized for different targets.

One interesting use of XQuery is to work with extensions that can support typical web server / servlet operations (request, response, session, etc.) to make XQuery act as a server "scripting" language in a manner to PHP, Perl, Ruby, Python, JSP, ASP.NET and so forth.

Another contender for such a query language is the W3C SPARQL language, which is designed to work with RDF-centric databases. It is possible that SPARQL may end up becoming dominant in this role within the next decade (as mechanisms for more structured Semantic Web capabilities come online), but it is almost certain that XQuery will serve that role in at least the short and intermediate term (See ROA and the Semantic Web, below).

Query capability is already a staple of SOA-based systems – SOA GET operations may differ somewhat in orientation (they are verb centered rather than noun centered, as ROA is) but in general they act to retrieve representations of resources (or links to representations of resources). ROA Query in general works on the assumption that a given resource collection

has an affiliated set of query operations that can be made on these resources that act as filters, returning a subset that satisfy a given criterion.

These queries are likely to be processed via XQuery or similar mechanisms (whereas SOA services will likely use a dedicated binary component) and the resulting sets may in turn end up being used in other pipelined queries or transformations. Indeed, one or the more intriguing specifications currently under development at the W3C is the XProc specification, which is design with the idea of creating "process" pipelines that take output from queries or XSLT transformations and passes them as input to other such processes.

This notion of pipelining is another characteristic that seems to emerge with ROA and is consonant with the idea that application development is heading towards the movement of streams of data (data in motion) through a series of pipelined processes in a manner that echoes pipeline flows in posix-based systems (Unix, Solaris, Linux, etc.).

Understanding the Philosophy of REST and ROA

In a number of distinct industry verticals, there are growing pockets of discontent about the evolution of SOA-based systems. SOA systems work well in environments where you have an existing component oriented infrastructure rendered in Java or various flavors of .NET, because these systems reflect the underlying paradigm of verb-oriented programming – in essence, SOAP messages act as serialized proxies for specific method invocations or their results. They treat the web as a giant processor, where each node within that network has distinct semantics.

This model of application emerged as a direct reflection of computer processing systems familiar to most programmers. It is conceptually

seductive to want to treat a remote server as being just another object in your environment with its own properties, methods and event handlers, albeit one connected by a comparatively long tether compared to internal objects. In some cases, where you have a relatively high degree of control (and trust) over the various nodes within the network, this approach can work well.

However, there are a number of situations that can occur where this approach generally doesn't work as effectively:

- Data models become sufficiently complex that they cannot be readily encapsulated in a small number of controls.
- Users need to interact with the application over heterogeneous platforms, devices and trust boundaries
- Data is provided through not only different data sources but also different types of data sources
- The application needs to work as well from a web browser as from dedicated, standalone applications.
- The underlying services stack within a vertical emerged before the SOA stack, and now represents a significant legacy system.
- The objects involved straddle the line between documents and data structures.

In most cases, these are all facets of the same underlying problem – there is a disconnect between the resource oriented world that has been part of the Internet from nearly its inception and the verb oriented world that reflects OOP-centric objects write large. Put another way: is the web an application … or a database?

This is not a minor question. Businesses, organizations, schools and governmental agencies have collectively spent hundreds of billions of dollars on software development across the web, with the underlying assumption being that the web was an application. Most of the application toolkits and frameworks in use today make this same assumption, and make

the further assumption that web development is basically the same as application development save that it is writ on a larger canvas. Hundreds of thousands of web applications are written each year, most heavily informed by this bias toward the verb, yet a surprisingly large number of those applications also fail every year because the complexities of design and architecture overwhelm the development team.

In some respects RESTful systems are not as capable as SOA-oriented ones. With a SOA-based system, you can ask a server to give you the exchange value of a certain amount of money in Euros vs. US Dollars based upon current Forex rates. A ROA system has no concept of money, Euros, Dollars, or foreign exchange rates, and certainly can't calculate anything. Instead, the ROA "system" here is in fact four distinct, albeit overlapping systems:

- One process pulls together the exchange rates into a single document.
- One process posts (publishes) the exchange document to a logical feed.
- One process queries the feed server to determine if the feed HEAD has changed, and load the new resources if it has into a local process.
- The final process then retrieves the relevant nodes in the incoming document and uses them as the basis for a calculation.

These processes are independent and asynchronous. The first and last step in fact have almost nothing to do with the publishing process (other than providing the input or consuming the output of same). Yet together these provide some insights into the Tao of RESTful systems:

- The web is made up of resources, collections of resources and links. That's all. Everything else is just window dressing.
- A resource is abstract – you can never see it directly, you can only see it through its representations.
- Resources are unique. The uniqueness is contained not in the content of the resource, but in its metadata. If five resources all point to the same content, they are still five distinct resources.

- The publishing paradigm is central to the web, and the publishing metadata of resources reflects that paradigm. Lose the metadata, and you've lost vital context.
- Computation is local, and is irrelevant to the web. The role of the web is to get you a representation of a resource – what you do when you get it is your own damn business.
- What exists beyond the URL is irrelevant – in a purely RESTful world, it does not matter if a resource is stored as a file, a database query, the result of a process or hieroglyphs carved on a wall.
- Resources are democratic;
- An Atom feed need not be a faithful representation of a resource's contents. Rather, it is a reasonably faithful representation of that resource itself.

We are moving into a realm where most data is addressable, and as such can be treated abstractly as resources, and where the cost of transporting resources is low enough that the idea of trying to work with modifying pieces of state remotely for efficiency or conceptual reasons no longer is a factor. The idea that you edit something by getting a copy of something, change the copy, then send the copy to replace the original may strike some as being wasteful of bandwidth, but validation is a powerful concept, and cannot occur unless you have the full context of the resource to work with.

Rectifying ROA and the Semantic Web

The Semantic Web is beginning to heat up in terms of interest on the part of developers, semioticians and vertical domain experts, and given the extreme emphasis covered in this paper on resources and resource oriented architectures, it may seem odd that there has been little mention of RDF or other aspects of the Semantic Web.

One of the most notable characteristic that can be found about ROAs is that they evolve and build on existing technology. Syndication feeds have existed in some form or other since the mid-1990s. XQuery evolved out of a

need to make HTML controls better, categorizations are simply an offshoot of microformats, and so forth.

The Resource Description Framework (RDF) is a remarkably powerful mechanism for describing not only resources but also relationships between those resources, and as such RDF documents might appear to be well suited for use as mediators within an ROA framework.

The problem that RDF faces in this context is that it carries a fairly high conceptual barrier to adoption within both the RDBMS and web development communities (especially the latter). RDF involves a process of conceptual normalization, reducing relationships to sets of what are known as RDF triples (containing a subject, an object and a relationship). This normalization can prove invaluable in the creation of certain types of databases, but such triples generally would need to be constructed by hand and would require sufficient understanding of RDF predicate logic in order to insure that what's created is in fact meaningful.

There will come a time in the not too distant future when these can be machine-inferred from the resource descriptions themselves, and can in turn be generated as web resources, at which point an RDF description becomes simply one more serialization format that an AtomPub service could provide or consume.

In the interim, it's pretty much inevitable that Semantic Web thinking will continue to affect and inform the deployment of ROAs, and over time (perhaps 6-8 years) its likely that ROA systems will be fully integrated with converging Semantic Web trends.

Serializing Resources

One aspect deliberately not covered here was the question about serialization formats, specifically the serialization of resource metadata. The web itself is a remarkably diverse environment, one in which preferences for given messaging and marshalling formats seem to change from one tech generation to the next.

A key assumption in the articulation of this model has been that processes such as data syndication through RESTful services can be custom tailored to fit individual needs without changing the underlying characteristics of the data model or its respect instances. For instance, its not hard to imagine, for a given resource such as books, that the book metadata's default schema will likely be Atom, as that schema corresponds very closely with the web's underlying publishing model.

However, one service may provide that Atom feed as an XML document, a second service may provide the feed as a JSON construct, the third may wrap a SOAP envelope around the relevant data, and a forth may encode this information as HTML or OOXML or PDFs. In the end, all of these are valid formats if you have both client and server components that can in fact take resources in that format and use them appropriately.

In general, the position taken in this paper is that an XML encoding offers more benefits in the long run than do most other formats, but so long as the other formats are essentially homologous with XML in terms of the underlying relational model, then they should do just fine in their own particular context.

Working with Atom and AtomPub

How many people does it take to write a newspaper? In the mid-19th century, as the cost of producing news broadsides dropped to a point where it was feasible to publish a paper weekly, then daily, this became a fairly

pressing issue. Up until then, most papers tended to have a staff of reporters that would be assigned to different beats, along with a collection of stringers, or independent journalists, who would write their observations, then sell them to a newspaper on a per story basis. Of course, the good ones tended to become full time employees at the larger papers, but this meant that smaller newspapers, with more limited budgets were often struggling to fill sections outside of their immediate area.

The rise of the telegraph in the 1870s changed that. A newspaper could take certain stories and telegraph them to smaller subscribing services, which would then either revise the story for local readers (if they had time) or would just take the stories as transcribed (more typically). These telegraphic subscription systems evolved into networks, known as *syndication networks*. In time, newspapers would pool their resources to create centralized syndication companies that specialized in certain areas of publishing – Associated Press (general news), Reuters (international finance) and King Features Syndicates (comics and related media), in essence becoming super-syndicators or *aggregators*.

What's important here is the model – individual writers would send their articles to a newspaper, which would in turn bundle these articles together as a package to a syndicator on a regular basis. The syndicator would then take the news feeds, and would filter them out by category, usually via ticker-tape (which used modified telegraphy) to give a short synopsis of the story. The subscriber would then read the ticker-tape to determine which stories had come in over the wire, and would telegraph (or later call) the syndication agency to send out the entire story (or more commonly would send out a block of related stories all at once, in order to make the accounting easier.)

REST and the Rise of Syndication Architectures

As it turns out, syndication is a very RESTful approach to publishing. A story is a resource – it can be turned into a distinct representation and it can be uniquely identified (by title, author, initial publication and date of publication). The act of publishing the actual content to the paper isn't necessarily significant, but the process of preparing a list of articles for syndication to the syndicator is – in essence, the syndicator can identify a given story as being a part of a certain newspaper, if attribution is otherwise required, but what the syndicator does, in essence is to prepare a catalog to a given *collection* of resources.

This catalog is in turn a *news feed*, which is also a reference-able resource. In some cases, the subscribing paper might request that the feed contain all of the stories that the syndicator can provide in a given category, if they have the bandwidth (and the money) to handle it; in other cases, the feed may just contain a short synopsis of the article and a reference link back to the original story in the catalog, allowing the subscriber to pick and choose (usually at a higher cost per piece, mind you) the articles they want for that day.

Although not a feature of the original syndications, there is one conceptual change that has direct applicability to the web. The syndication catalog was not updated continuously. Instead, the syndicator would typically set a deadline then would transmit the catalog in batch (because that was most efficient given the communication channels of the time). However, once the efficiency of the channel improves to a certain point, it becomes far more effective to simply post the catalog as a resource and have subscribers retrieve the catalog when they need it, as well as providing a way of identifying when a new catalog has been posted. If between two calls the catalog remains unchanged, then there's no need for the subscriber to retrieve another copy of the catalog (or in contemporary parlance, if a ping returns an identical ETAG for a feed resource from a previous ping, then the feed is unchanged).

In this system, there are very few verbs, and they all relate to the publishing of the resource, not the resource itself. Put another way, the semantic content of the resource is not part of the syndication process, except as how it affects the content of the summaries and the retrieval mechanism. This *semantic neutrality* is a very important aspect of REST, especially compared to SOAP based RPCs. Semantically neutral systems in general have a low degree of coupling with other systems, simpler interfaces, and as a consequence considerably wider applicability. These are all highly desirable characteristics on the web.

An Atom Anatomy

The history of computer syndication feeds is one of the more exciting in the field, with heroes and villains, fortuitous discoveries and disastrous decisions, great hype and greater despair. Unfortunately, with the exception of understanding that there are two active feed standards, it's not terribly germane to this paper, so will be tossed out.

At the present time there are two newsfeed standards – RSS 2.0 and Atom 1.0. Despite the versioning, RSS 2.0 is the older specification, was geared almost exclusively for blogs, supports XML poorly, has at best a very sketchy definition (and no formal schema), and as such is likely to fade away over time, though that time may be measured in years yet. It's unlikely in the extreme that RSS 2.0 will end up becoming central in the realm of XML data publishing, by most indications, and as such, will not be discussed further in this analysis.whe

In 2003, after the prolonged debate about RSS had reached a head with the creation almost simultaneously of RSS 1.0 and 2.0 as completely different, incompatible specification, Sam Ruby had had enough. A web architect working for IBM, Sam posted a notice to the xml-dev newsgroup asking what actually would make for a good syndication format. When several

people wrote back lamenting the current state of affairs, he set up a small newsgroup specifically dedicated toward improving syndication.
In time, a number of quite prominent people in the XML community joined up, including Tim Bray (of Sun), who was the editor for the original XML specification, and they worked diligently on a specification that started out life as Echo, but in time was changed to Atom.

In 2005, the Atom syndication format itself was completed, and was then published by the Internet Engineering Task Force (IETF) and submitted as a note to the W3C (giving it a w3c namespace designation).

A typical Atom feed is an XML document that has two distinct sets of elements. The outermost `<feed>` element contains information about the feed itself, including:

- A unique identifier for the feed
- A title for the feed
- Which agency produced the feed
- When the feed was created and last updated
- The link to the URL that the feed came from (as well as any additional links to other resources as appropriate).
- The author(s) of the feed
- One or more categories that provide taxonomic context for the feed.

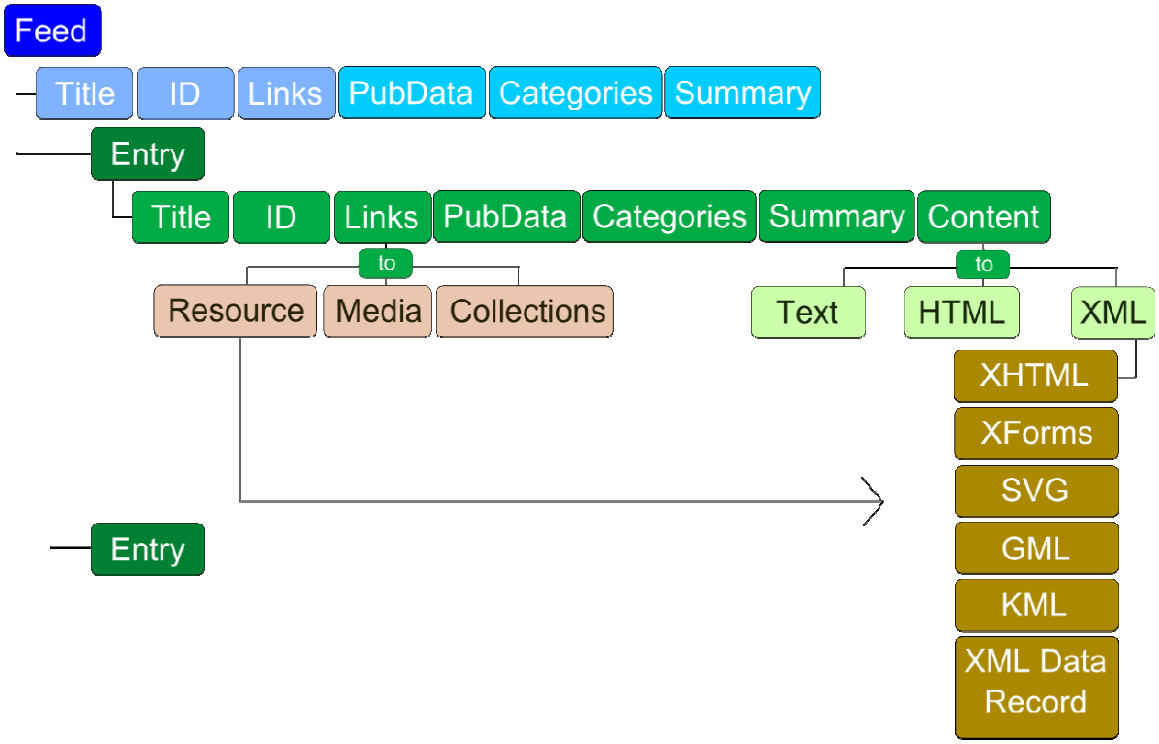

FIGURE 1. ATOM FEED BREAKOUT

Each <entry> in turn provide multiple links and descriptions of a given resource (see **Error! Reference source not found.**). Each entry has an identifier (often, but not always, either a URL or a universally unique ID (UUID or GUID), a title that's used by news readers, publishing information for the resource (when it was created and last updated, who it was published by, and similar data), a category field which can also have a formal taxonomy identified for the term, and a summary. Additionally, the entry has one or more links – to the "record of reference" for the entry that shows where the source is, to media that may be used by the resource, perhaps to different versions of the entry, and so one. The "alternate" link is perhaps the most significant for blogging, because that references the

"source" document (and as such the one used by news feed viewers), but other link types can be defined.

The final aspect of an entry that has a direct bearing on data systems is the `<content>` element. The content of an entry does not *necessarily* contain a complete representation of the whole resource. In a blog, for instance, it is not at all uncommon for the content block to hold a teaser, which might hold a couple of paragraphs of the blog to entice readers to follow the link back to the full blog posting.

On the other hand, if that resource in question was a resume, then the content block might actually contain a "resume-lite" as an XML document which featured the owner's name, position sought, basic skill set and the current or most recent job that the person had. This light-weight version is in turn useful as a way to build tables or lists via XSLT (or XForms) of elements from the feed, again with just enough information to fill out a table without requiring a potentially much larger download of the entire data record. This becomes especially significant for large "data-documents" such as aircraft manifest specifications, financial statements and other similar entities.

Moreover, the content block plays an additional role as the payload for AtomPub messages, covered in the next section.

One final aspect of Atom is worth touching on. Unlike most other RSS feeds (except RSS 1.0), Atom has a namespace. The specification also allows for the use of adding alternative elements and attributes to an Atom feed, so long as they are given in a different namespace. This strategy is used with the geoRSS format, where geographical information is added to Atom, and is also used extensively as part of Google's GData formats.

Being Resourceful with AtomPub

One of the things that differentiated Atom from other RSS specifications was the realization on the part of its creators that the syndication format by itself was only half of the story. Most blogging server applications established specific APIs (usually XML-RPC based) for posting new content to the server, for retrieving content, and for handling other publishing operations. Such applications included BlogSpot (later Google Blogger), Six Apart's Movable Type, WordPress, and a number of others.

The Atom group, however, recognized that one of the central challenges to working with syndication was that a syndication format by itself was not enough. By taking advantage of a strong resource basis for the architecture, a clearly defined publishing protocol built with REST principles could revolutionize not just blogging but data publishing in general. This particular protocol was formalized as a specification with the IETF (IETF

RFC 5023[1]) and given the rather unwieldy moniker Atom Publishing Protocol, originally abbreviated as APP, though more recent usage within the working group itself suggest that AtomPub has become the preferred "short name". Joe Gregario of Google and Bill de hOra of Newbay Software are the editors of the specification, and two of the most vocal evangelists for the technology.

Why a protocol rather than an application? In this case, it helps to define exactly what is meant by a protocol. A *protocol* is a set of conventions or rules that describe the process of communication – in essence, it's the rules of a conversation. For AtomPub, the protocol was intended to reinforce Fielding's REST philosophy, and as such was established with HTTP as its model (and basis). Moreover, by dictating how conversations were made (or, put another way, by setting up a standardized programming interface between the user's machine and the AtomPub server) the working group

[1] http://tools.ietf.org/html/rfc5023

made it possible for anyone to implement an AtomPub server without any specific reference to the underlying technology.

This strategy appears to have been a sound one. Tim Bray of Sun has created an AtomPub mod for Apache[2], AtomPub features prominently in the open-source eXist XML database server[3], and Microsoft is incorporating AtomPub into their Live Platform server[4]. AtomPub support has also been added to the Django Python framework[5], Movable Type supports AtomPub

---

[2] http://www.tbray.org/ongoing/When/200x/2007/06/25/mod_atom
[3] http://exist.sourceforge.net/atompub.html
[4] http://www.25hoursaday.com/weblog/2008/02/28/WindowsLivePlatformNewsMicrosoftStandardizesOnAtomPubForWebServicesAndOtherStories.aspx
[5] http://code.google.com/p/django-atompub/

in addition to its own internal formats for publishing[6], and Google's GData architecture is being refactored with AtomPub as a core technology[7]. Given that the AtomPub specification was only released in October 2007, this should indicate how significant AtomPub is seen throughout the industry.

For all this, though, it is worth asking why AtomPub has turned heads in both the web community and at a higher level. A big part of the answer has to do with the growing awareness about the degree to which RESTful principles underlie the web – and the corresponding awareness of the importance of collections within that view. Especially as XML becomes more prevalent as a storage format for not just traditional *documents* but also for more data-centric entities, developers have begun realizing the benefits of building abstract "folders" that are date-ordered as a default (on the

---

[6] http://www.movabletype.org/2007/12/atompub_support_and_new_edit_a.html

[7] http://googledataapis.blogspot.com/2007/11/atompub-interop.html

legitimate basis that what is newer is also more likely to be that which the user has the greater interest in for either reading or editing), particularly when you can combine this with a resource-centered approach where each container represents a class of similar objects.
As a brief aside, consider how files are traditionally kept in folders on a file system. The user will typically create their own folders, often based upon very immediate or local considerations, will then place all working files potentially germane to the topic of each folder (as well as many files that end up their largely by accident) in this folder, with various file types all mixed together. With small hard drives, this was typically a manageable arrangement. However, with hard dives approaching terabyte size, it is likely that most users are typically aware of at best 1% to 2% of all the files on their systems, to the extent that search engines have increasingly become the most dominant navigational tool not just on the web, but for one's on hard drive.

Contrast this with the way that an AtomPub server is conceptually organized. AtomPub breaks up resources by resource type – all "blogs" in one blog folder, all "spreadsheets" in one spreadsheet folder, all vector drawings in one drawings folder, and so forth, in all cases with the default (though not only) organization being that the most recent (and hence likely most relevant) resources near the top of the stack. What's more, the publishing metadata about each resource is an atom entry with its own title, summary, owner, publication date and categories fields, *each of which are distinct from the actual resource itself.*

AtomPub has two distinct types of tracking documents that can be queried. The first is the *Service Document*, which provides a listing of all of the various collections (think document types) that the system currently hosts, as an XML document. This makes it possible to get a snapshot view of exactly what kind of document functionality currently exists on the system, not by the file type (which in most cases will be XML) but by "namespace" type.

For instance, suppose that you run a coffee-shop cum music-shop, and you maintain four separate namespaces describing the inventory state of drinks made, food items, music CDs, and merchandise. In an AtomPub view of the world, each of these are separate collections, even though internally the representation may (or may not, for that matter) be XML documents.

The second tracking document AtomPub uses is the *Category Document*, which takes advantage of the atom:category property. Some categories may be service type specific – you can break up the coffees sold into the categories of lattes, mochas, cappuccinos, hot chocolates, teas, and so forth, while others may work across services – such as "regular sale price","spring promotional sale", "barista personal discount" and so forth. The category document thus contains all of the categories that are currently defined on the system, and can additionally include for each category a category schema that's usually expressed as a namespace – "urn:coffeetypes" or "http://www.mycoffeeshop.com/xmlns/salestypes". A given entry can have zero or more categories assigned to it.

From an organizational standpoint, the use of service type and category type together with the normal date ordered arrangement provides a natural navigational mechanism that's easy to implement but powerful in its ability to locate information, especially as category types can be indexed for fast access. For instance, retrieving all drink or food sales but not general merchandise sales during the spring promotional sale can be done without ever even referencing the underlying data documents – they are contained within the Atom entries (which indeed may only contain the relevant records via reference as a link).

Additionally, the Services document is broken down into workspaces, which are generalized containers that are deliberately kept rather ambiguous within the specification itself. A workspace contains a tag identifying one or more collections, but again this is more of a categorization

mechanism, as a services document may contain more than one workspace, each of which may in turn contain a reference to the same collection – in essence it's a category at the collection level, rather than the entry level, and exists primarily as a hook for handling workflow.

The ROA Market Landscape

The changes that are occurring with resource oriented architectures are happening because of several factors. Most of these factors are longer time and are generally not tied into a specific product release or marketing strategies. As a consequence they should be seen as either shaping or being shaped by broader IT or social issues.

XML has become pervasive within most organizations, due either to governmental mandates, industry consortia standards, documentation or workflow management systems, or some combination of these.

Pressure is increasing to make organizational data available to users in as broad a manner as possible, and that increasingly means syndication feeds.

Blogging infrastructures, used increasingly as a communication tool within organizational environments as well as externally to the organization, are paving the way for feed-based design elsewhere in the system.

A number of key standards (XForms, XQuery, Atom, AJAX, XSLT2, etc.) are reaching or have reached completion within the last couple of years, and these particular standards work well together in a RESTful context.

The rise of AJAX has made it possible to implement consistent RESTful clients on most browsers, and has made it possible for alternative approaches to be run constructively on browsers that do not have native support for RESTful systems.

SOA based systems have generally raised the awareness of distributed XML within organizations, but SOA has in general failed to cross the last mile into user's systems. ROA offers up an alternative that people are beginning to explore.

The recent establishment of OOXML and ODF have opened up the possibilities of including *all* office documents – including spreadsheets, presentations, internal graphics and so forth, as part of a generalized XML workflow. Significantly many XML databases and XQuery systems include the ability to work with zipped resources, meaning that it is possible to "publish" such documents to an AtomPub server and then make them queryable and manipulatable.

One of the things that has struck this writer as he put together this paper was the degree to which this particular architecture is being simultaneously "discovered" in nearly identical forms in very different industries. From library science to newscasts, health care systems to telecommunications, IT managers and software developers who have been struggling with the "last mile" problem of getting data to and from users have looked especially hard at syndication services as a way of bridging that gap.

Most of the technologies covered in this paper have been around for some time, but they have taken a long time to reach a level of maturity necessary to support ROA as outlined here. Significantly, by September 2008, barring any last minute objections that currently seem unlikely, all of the technologies covered here will have at a minimum fully endorsed specifications, and most will in fact have second or even third generation support for these technologies.

This paper has deliberately steered away from citing any particular vendor or technology (this will be done in a subsequent set of papers), but on each

front there are multiple participants and usually an emerging leader with the space.

IBM has recently unveiled a nearly complete ROA stack, and has support for an XQuery engine, a couple of XForms implementations and much of the stack. Not insignificantly, a number of the key engineers who contributed to the Atom and AtomPub specifications are senior architects with IBM, including Sam Ruby and Joe Gregario.

Google's GData format was one of the first to effectively recognize the power of Atom feeds as a formal mechanism for data syndication, and most of their current web services API is much more ROA focused than SOA based. They are similarly working on AtomPub based systems as well. This author is also inquiring whether such support will include AtomPub support within Google Gears, as this would provide a powerful "offline" support for the language.
Yahoo Pipes provides a way of creating syndication best "widgets" through a drag and drop type interface (http://pipes.yahoo.com/pipes/). Indeed, spend some time studying Yahoo pipes, as it illustrates the power of syndication services to do a variety of "real-world" tasks.

Microsoft has recently announced that they will be supporting AtomPub syndication with a number of their Live services, and Atom and AtomPub are being explored as vectors for the next generation of SQL Server. Given the connection that ROA has with categorization and semantic web issues, parts of this stack as well are being examined for possible use in achieving a generalized virtual file services implementation.

The Mozilla Firefox browser has had an ongoing XForms component supported for a couple of years, which is now being extended to include the XForms 1.1 specification. FormsPlayer remains one of the best plug-in modules for extending XForms support into Internet Explorer, and both

Orbeon and Chiba have managed to gain significant market share as integrated browser / server cross implementations of XForms, with Orbeon having the more advanced product.

Finally, the XQuery marketplace has seen major activity just within the last year. IBM, Oracle and Microsoft all have some level of integrated XQuery support, though IBM has probably the best implementation of the three with the IBM DB2 XML Database. Microsoft's implementation relies upon an older version of the specification. Oracle has recently announced that they are embedded their XQuilla XML database into the open source Oracle Berkeley DB XML database (formerly Sleepycat), and recent press announcements indicate that they are investing significantly in updating that technology in light of the growing popularity of XQuery systems.

Similarly, the MarkLogic XML Database has been gaining traction on both the publishing vertical (which was an early adopter of XML Database technology for document storage) and increasingly in general data applications. On the open source side, the eight year old eXist database (this author's personal favorite database) is rapidly becoming for XQuery systems what mySQL has been for SQL systems – the quiet back-end server of XML content. Both systems, not surprisingly, work well with the ROA stack.

Currently JustSystems, which has long had a strong presence in Japan but a considerably smaller footprint here, is perhaps closest to building an integrated ROA stack with their *xfy* systems, and their presence in areas such as health services (with HL7) and business reporting (XBRL) make them players to watch in this space. IBM is likely to be their strongest competitor in this space.

Given that in many cases these vendors developed their respective implementations looking not at long term integration but rather immediate

customer business needs means that they are continuing a trend that has persisted for some time – ROA systems are evolutionary rather than revolutionary, becoming obvious as an integration strategy only after the various underlying technologies come to fruition. Such systems tend to follow logistic growth patterns where much of the development occurs in isolation and below the radar until a critical cusp point is reached, at which stage technological adoption grows swiftly.

Even with current economic headwinds, it is likely that ROA system growth will become noticeable by late fall 2008, and will become a dominant theme for coverage through much of 2009 and into 2010 before it goes through a slowing period by spring 2010.

Recommendations

The ROA stack is still emerging, and as such it's unlikely that you will be able to hop out to the store and get REST In A Box (bed sheets included!) any time soon. However, there are a number of things that you can do for yourself and your organization that will make transition to a ROA (or more likely mixed ROA / SOA) approach easier.

Identify the critical collections of resources that you currently work with in your organization and ask whether a ROA approach can be more readily utilized for making them available in your organization. In particular, look at the new generation of XML documents (including ODF and OOXML, as well as DITA, DocBook, TEI and of course XHTML) or industry vertical taxonomies (XBRL, HL7, GML, etc.) as appropriate for your organization.

Set up internal AtomPub "blogging" systems in your organization, both to familiarize people with syndication services (if they are not already there) and to establish the necessary conduits that can be later expanded to include XML data publishing.

Download and set up one or more XQuery-enabled XML database systems – this author recommends the eXist database (http://exist.sourceforge.net) as a relatively painless introduction to XQuery, though most other XML Database vendors include free evaluation or developer versions.

If one does not already exist, designate someone in your organization as an XML Resource Administrator. This person (these people) would both be responsible for maintaining the XML resources within an organization and would handle the creation of XML databases or bindings for similar repositories. The best candidate there is a person with extensive XML and AJAX experience, especially working with web services and data streams, rather than a SQL DBA.

Similarly, tap someone with XML modeling and AJAX development skills to start working with XForms and related XML RIA technologies. Both XQuery and XForms can have a fairly steep initial learning curve – six months or so in both cases, though the basics can be learned fairly quickly.

This is a technology that should be thoroughly tested through multiple pilot projects that would focus on building an ROA based system that will put these resources into the hands of both internal and external users to your organization. Remember that at the end of the day you are still dealing with syndication feeds, which means that such projects may very well make your data available to subscribers with the appropriate security permissions via a newsfeed on a Google home page or Yahoo pipes.

A number of industry consortia have invested heavily in ontologies and alternative web services stacks that are heavily SOA-influenced. If you are involved with such efforts, take a close look at AtomPub and ROA as an alternative approach that is still standards based, generally offers more

flexibility in terms of the underlying model, and works well with the resource bias that most such organizations have.

Finally, keep abreast of resource oriented architectures in the technical media. Currently it has a comparatively low profile, but this should change by the end of 2008.

The Details

There are a large number of concepts that are likely to be new to readers within this paper. Much of the formal analysis deliberately focused on these technologies at a very high level. The content within this section of the paper will look instead at individual implementations of a ROA stack built upon much of the W3C technologies stack.

An Introduction to Syndication Services

Ask yourself a simple question: where did you get your news today? Chances are good that your first response wasn't a newspaper, television or radio. Increasingly, your news channels are coming either from a web site showing syndicated content, an email alert or newsletter or a news "reader". Moreover, in most of those cases, the story titles and teasers spent a certain amount of time as a syndication feed.

Syndication can be thought of as an invisible revolution – an idea that is so obvious, that has spread so quickly, few people really have an appreciation for just how profound the revolution has been. Syndication feeds, otherwise known as news feeds, have become

the nerve impulses of the information society. They provide the latest headlines on your favorite news portal or notify you when your favorite blogger has just posted a new story. They're playlists of your favorite songs or collections of YouTube videos on the latest breaking financial news. Request a list of hotels in an area from a mapping site, and you're accessing a news feed. Checking your appointments from a web-based calendar? It's a newsfeed.

A syndicated feed, in its most basic form, can be thought of as a small library card catalog shelf, say Science Fiction Titles. That shelf has a definition location (3rd in from the left, 2nd from the top in the entire card catalog) and thus has an address (3,2) and a name (Fd to Fe).

The shelf in turn contains library cards. A library card is not a book, or a video, or a record. Instead, it is a description of a book, a video or record, such as a novel. It contains the primary name of the book ("Foundation"), the author ("Asimov, Isaac"), an ID (such as an ISBN number, or even more likely, a scan code) , a short description or abstract ("First book of the Foundation Series, in which Harry Seldon predicts the demise and rise of civilization in the galaxy."), when the book was first published and most recently updated, a category that describes the book in some schema ("scifi") and most importantly, where the book is located (one or more call numbers).

Most syndicated feeds follow nearly this same structure, though the specific terms used to describe them may be different, details which will be explored in greater depth in this paper. Yet one of the most significant things to note here is that such a library card is ultimate a reference to the resource in question, not the resource itself. The library card is not the book.

Take this concept a step further. The book's location is a link, though in the case of most libraries it is a relative link that's only appropriate to the library

system in which it's found. Put another way, the link within the library card entry is an address into the universe represented by the library (avoiding for the moment the whole concept of inter-library loan, where the metaphor breaks down rather badly). However, suppose that the library in question was global in question, such that the call numbers that were given were ones that uniquely identified that book anywhere in the world.

There's another abstraction here that's perhaps even more subtle (and more important). Suppose for the moment that someone decided to digitize the contents of the book in question, such that, when you requested a book from the library by its call number, rather than sending the actual book, what the library does is ask for a digital copy to be sent, and the library then creates a reproduction of the book (with lesser or greater fidelity) and lets you have that.

Indeed, this creates one of those interesting little conundrums that seems to crop up whenever dealing with the web. The call number doesn't actually point to the physical resource – it points to a process that will return a representation of that resource (indeed, the book as such may, once reproduced, be shredded and destroyed – a concept that was explored in some depth by science fiction visionary Vernor Vinge in the book Rainbow's End – or archived.

This is exactly what happens on the web. The call number (or URL) points not to a physical resource, but rather a process that will transfer a representation of that resource to you when you make a request against it. Such representational state transfer has come to be known as REST, and it can be summarized simply: *REST is the process whereby a representation of a resource's current state is transferred from one point to another on the web.*

For the most part, the books in the library are static … you do not generally expect a given book to change from one visit to the next. Suppose, however,

you're talking about magazines. You go to the library and ask for the latest issue of Wired. So long as it's still being published, if you go once a month you'll likely find that "the latest issue of Wired" – which is definitely a resource – will be different every time.

Indeed, it's possible to envision the magazine as a second "feed", where the resource in question is itself a table of contents and each entry in that table of contents points to an individual story or illustration, perhaps with just a small excerpt, an author, and an address that consists of the page within the magazine where the story starts. This idea of feeds point to feeds is something that seems to emerge spontaneously when you deal with such an architecture.

Thus the first resource (the latest issue of Wired) certainly has a physical representation, but it also has a conceptual one that's dynamic in nature. You can talk about having Wired magazine in the library, and what you are referring to is actually a collection of separate objects (each issue of the magazine). In general, though, most of the people who come to the library to read Wired are not going to be looking for a specific issue (May 2008) but rather are going to be most interested in the *newest* issue. Because of this, the concept of a news feed is also typically associated with the aspect of novelty – ordering the resources on the basis of how recently they were published.

Because we're dealing with a process generated representation of a resource (Wired magazine, for instance), one effect of this process is that it how the magazine gets to its call number is irrelevant. Someone may have scanned each page of the paper resource, then run a process to convert the text into meaningful layout. Someone may have stored each story as a separate file that was never paper at all. Perhaps there's even a room full of monkeys that are generating the stories so referenced randomly (this is not an indictment of the editorial quality of Wired, by the way). The point is that regardless of the method of production, the resource will be the same in all cases – the call

number (URL) is an abstraction that hides the details of the origin of the stories.

Of course, one particular approach that could be used is for the writers of each of the stories to produce them on their own word processors then send the stories with just enough information so that the editors can make sure they have the right author and where the original story comes from (so they can communicate with the writers in case something goes wrong). The editors will then fill out the publishing data not in the story itself (the reader's not interested in that particular information, at least in its raw form) but in the electronic version of their own card catalogs.

On the web this particular process is known as *blogging*. The term originally came from "web-logging" as coined by (?), and this concept actually also has some interesting ramifications. A log is analogous to a journal – you add an entry into the log at the bottom of the page after all other entries, such that over time you create a chronological collection of entries. Again, in general what is most important is that which is most recent, unless you are in a situation where you are looking for information within the log (the ship just sank, and you want to know the reasons leading up to the sinking).

Note one of the advantages of a log, though. Each entry is "time-stamped" and also may have some other ID associated with it. A news feed could be generated automatically by providing the address of each entry in the log in reverse order by time, and in fact this almost exactly how such news feeds are created.

However, you could also impose a different ordering based upon some other criteria. For instance, you could create a collection of stories authored by a given person or people – say writers specifically who write for Wired. You could create an anthology by collecting all stories that have to do with robots as a category (this category having been one of the things that the

editor assigned when she built her library card of metadata). You could create such a collection based upon a peer-review rating, such that only those stories that exceeded the rating would be published. You could even combine all of these, as separate filters to get the best stories written in the last year about robots by writers featured in Wired.

And therein lies something … fantastic.

How to Publish a Robot

Suppose that you are in charge of the Foundation Robot Company's documentation department. Rather than dealing with stories or posts about political figures, your job was to manage the technical documentation about each new robot model that your company created.
Now, technical documentation is generally considerably more structured than a blog – you typically have specific models and model lines, assemblies and interfaces, hardware components and software components. Indeed, because each specification also needs to go out to your subcontractor (and in some cases will be developed by those subcontractors), you need to spend some time developing a formal model of this documentation.

For sake of argument, assume that this model is expressed in XML (it doesn't have to be, but XML's long been associated with documents, and there's a fair amount of information that seems to be more data-like than it is prosody). You go out and commission your IT department to create a special "editor" that lets you build the XML (bypassing the question of what exactly that editor is, for the moment) from a special template.

When the robot document specialist finishes up the document, what if, rather than saving it as a file, he sends it to a web-log of robot specification documents, adding in a few pieces of categorization metadata that describes

which model the spec is for? The interesting thing here is that the weblog "server" functions in exactly the same way as above – it adds the specification itself to a log and creates a new metadata record that points specifically to the log itself, and assigns to that record an ID.

Now, this is where things get interesting. Suppose that someone wants to retrieve that particular documentation. In traditional programming, this is where the programmer starts to sweat. You see, there's no filename. To get to the document, instead, you have to ask the system to give you a list of all of the robot specification documents, which it returns as (you guessed it) a news feed.

Of course, one particular category that you may have included in the metadata is the robot model, which means that if you know the robot model, you can retrieve any specifications on that model. Note here that you're still dealing with a collection – there may be multiple specifications for model "R2D2", for instance, but these specifications are returned in reverse chronological order.

Keep in mind, however, that all you are retrieving is publishing metadata, not the actual document, and that publishing metadata as a consequence is considerably more compact than having to do an indexed search by keyword. Once you have a given entry, you can then retrieve the appropriate data (the specification) by its address. This means that you move from a single search criterion (a title, usually arranged in alphabetical order, which is what a file system normally gives you) to establishing collections using multiple dimensions of categorization.

If this sounds more like a new kind of file system than it does a news feed, well, it's because one of the implications of syndication feeds is that categorization lets you change the nature of how you organize your data. Indeed, when you think about the fact that many hard drives are now

approaching (and exceeding) the terabyte size, with the resulting potential of having hundreds of thousands of files, the idea of being able to do categorization based “virtual” folders is becoming not only feasible but increasingly necessary. This will be covered in greater depth in The Details for this paper.

There is, however, another subtle shift in the way that a feeds-based architecture changes the way that you think about information. For instance, going back to the robot specification for a moment, consider that you have three distinct groups contributing to the specification: hardware, software and project management. The specification for hardware will look at what each piece of hardware is and where it fits together with everything else, the specification for software will contain all of the codes for operating the robot, while project management details which vendors are supplying various pieces or services.

If the specification has a special viewing or editing application, trying to build a single one for all three different aspects could be troublesome … far better to break the specification into three pieces and build the editing applications for each. As a consequence, the formal “specification” in turn consists of detailed specs for each area and a fourth document, a manifest that provides links to the other three as well as perhaps just enough overview material to explain what’s going on. Put a different way, by decomposing the specification, you make it modular.

Again that manifest looks a great deal like a table of contents for a magazine – or if you treat the introductory material as its own standalone section, the manifest happens to strongly resemble a syndication feed, though in this case the feed acts as an index into different document resources. The primary difference between the manifest and a news feed is that the ordering in the latter is determined by an automatic process (the order of

publication dates) while the ordering in the former is determined by editorial judgment, but they are otherwise the same structure.

For the most part, order criterion should thus be considered a property of all syndication feeds. Furthermore, once you do establish the ordering, it also becomes necessary to establish "paging" – how many entries appear within a given feed. In a simple reverse chronological feed, that paging can be determined simply by fiat – the most recent twenty records are given first, then the next twenty, then the third, until you return all of the records. In an editorial feed, the paging is usually determined manually, perhaps by some external criterion (retrieve the latest of each of the introduction, hardware, software, and operations documents and combine them in a given sequence).

Both ordering and paging are operations that are associated with such syndication feeds as a class. Filtering is another operation – restricting the set of all items under consideration to only those that satisfy a given criterion. All of these operate on collections of data, rather than on single entries … and they are operations that are traditionally seen more often in data management than they are in traditional document management.

It's worth restating this point:
A syndication system must be able to filter a data set by one or more criteria (**filtering**), sort that filtered set by one or more criteria (**sorting**) and then partition the resulting set into discrete pages of content based upon some criteria (**paging**), regardless of the nature of the data. The result of these operations is then serialized as a feed.

Note that somewhere along the line a miraculous transformation occurred: documents dropped out of the equation. Each entry in a feed is a link to a resource, but there's nothing there that states that the resource must be a blog of marked up HTML. In the case of the robot specifications, the

resources involved combine prosody (text elements used for traditional publishing) with other combined pieces of information (such as technical parts lists or costs).

Awaking to REST

One of the fundamental concepts of the web is the notion of resources. A *resource* on the web is a block of content that can be addressed using a *universal resource locator,* or URL. The HTML of a web page is a resource, the images files that are pulled in via <img> tags are resources, as are the JavaScript files that are imported as part of the web page.

It's tempting to equate a resource with a file, but that description doesn't always hold true. More appropriate, perhaps, is the notion of a resource as something that can be streamed to the user – thus, a web service that uses a GET method to retrieve it is a resource, although this distinction can occasionally get lost in the broader role of web services. Significantly, what is getting streamed is not in and of itself the resource. Rather, the resource is an abstract object, while what gets streamed is a *representation* of that object.

The protocol used to access these resources is known as the Hypertext Transport Protocol, know far more readily by its acronym: HTTP. Most people know HTTP from the seven character *protocol declaration* at the beginning of nearly all URLs – `http://`. Yet this familiarity is a little deceptive. Even seasoned web developers tend to misuse HTTP compared to the way it was originally intended to work, and they underestimate the power of the protocol because their familiarity extends only to rather jury-rigged constructs that are in many ways accidental artifacts of their design rather than understanding the protocol itself.

For instance, when you pull up a web page, what is sent is the HTML code that represents the web page, that can be used to create a local instance of that page within the browser. Every time you make a request from the

resource via its URL, you will always get a representation of that page. Note that there is nothing that says that what gets sent back is the same content every time, mind you … indeed, on a busy news site, if that URL always returned the same content, it wouldn't be news any more (perhaps it'd be olds), but what is significant is that the resource "My News Site" is conceptually the same every time an HTTP GET command is sent to the URL – only its immediate representation changes over time.

So far, this mirrors the familiar way that the web works. However, there is considerably more to HTTP that's outside of the normal range of experience even of software developers. For instance, with the PUT method, you can send content to a specific URL and, if you have permissions access to doing so, have that content be updated the resource if something already exists there or create a resource based upon that content if it doesn't.

The advantage to this type of behavior seems so obvious (no more FTP, nor more SSH or any of the dozens of other seemingly awkward, amodal ways currently used to populate websites) that it may seem surprising that so few web servers actually support PUT. Part of this is historical accident – a number of companies in the mid-1990s didn't see the distinction between PUT and that other HTTP stalwart, POST, when developing either web servers or web browsers – and part of it had to do with the perceived security risks to the web at a time when web authentication and security was still very primitive.

POST, in turn, became the preferred mechanism for sending content to the server, because you can pass property data that extended beyond the then known limitation of 255 characters that GET suffered. However, POST as originally defined was intended as a way of adding a new resource to a collection of resources (a collection of course being a kind of super-resource of its own).

To put this in database terms, GET would be used to retrieve a record with a known identifier, PUT would update a record with a known id with new data, while POST would add a new record to the record collection, assigning it an ID as part of the process. Two more HTTP methods – DELETE and HEAD – complete this set, with DELETE returning a specific resource as specified by its URL and HEAD returning a collection of "headers" about the resource in question, including its creation date, its last update date, its internal file representation if appropriate and so forth. Note that these five commands sound very much like what you would expect of any record-oriented database, even if the record in question is a more ambiguously defined document.

There is one more aspect of HTTP systems that's implied, but is nonetheless extremely important to the idea of XML data publishing … the notion of the *named collection*. Most people understand folders or directories, because these structures are found in the operating systems of most contemporary personal computers and servers. A named collection can be a folder holding files, but there's nothing that specifically requires that within HTTP; indeed, a named collection can also describe a record-set holding a collection of records in a database, or it can be sibling elements in an XML files, or sibling XML trees, otherwise known as a *grove*, in an XML database, or any other sequential set of items. The only overriding characteristic of a collection is that each child resource in that collection be able to be addressed uniquely.

Roy Fielding began his own experiences with REST when working with the NCSA server originally developed at the University of Illinois. As a graduate student (and eventually a doctoral candidate) at University of California, Irvine Campus, Fielding discovered the NCSA server project that had laid fallow for some time and he decided to work on as part of his Master's degree. He in turn farmed out maintenance of the server to other developers both at the school (and in time elsewhere), setting it up on a modular basis with each module being considered a patch to the original

application. After a while, the team realized that there were more patches than there were original applications, and they rechristened it the Apache (a patchy) server, which would go on to become the dominant web server on the planet.

In 1999, Fielding wrote his doctoral thesis on both Apache and the web in general, describing many of the characteristics that both defined the way that the web itself worked and focusing on the document publishing aspect of it. Within this thesis (which eventually garnered him the PhD) he also coined a term that's become central to contemporary web methodologies. Given the abstract resources that made up the web, what was ultimately more important were the *representational states* of these resources and the way that they were *transferred* from one node to the next. He dubbed this concept *Representational State Transfer*, abbreviated as REST.

A REST-based (or RESTful) system is thus any system which uses URLs that explicitly describe resources in a collection aware fashion, that transfers state back and forth as discrete data bundles, and that make use of the underlying hypertext linking that relate two or more resources with one another.

Note that such a system can be thought of primarily as resource, or *noun*, oriented. This differs significantly in focus from the way that many contemporary web services are defined. In that particular case, the web service is much more *verb* oriented – a URL's intend (and hence structure) is intended to perform an action, and as such the URL can be thought of more as a function with multiple parameters. SOAP-*based remote procedure calls* (RPCs) take this notion one step further by placing the verbs within a messaging envelope; the recipient URL in that case is a server that acts more like an OOP object, and the HTTP role diminishes considerably – its primary purpose is simply to act as a conduit to a service, and it is that service, rather than the HTTP underlying it, that does the actual processing.

In many cases, such SOAP-RPC systems make a great deal of sense, especially in environments where transactional integrity is paramount. However, such services oriented architectures (SOAs) can also impose a significant conceptual and computational overhead on building web-based applications, especially when compared to REST-based alternatives. One goal of this paper is to explore at least one such alternative, and show its strengths and weakness compared to the more "traditional" SOA architecture.

Atom and AtomPub

The Mechanics of AtomPub Publishing

One of the more remarkable aspects of AtomPub publishing is the fact that it is surprisingly simple, especially with what's available within a web browser. Indeed, for the most part, AtomPub is just a generalization of the normal REST mechanism that has been in place for retrieving content from the web since its inception.

There are three design principles that shape the nature of AtomPub services. The first is that, beyond the minimum actions specified as part of a given AtomPub service, *the AtomPub protocol does not specifically preclude any other processing behavior*. What that means in practice is that post or put actions might update other records, validate the record against a schema, send out a notification or some other action and still be within compliance of AtomPub behavior.

The second design principle concerns URIs: the specific URIs for covering given actions are not explicitly stated in the specification. Rather, it is the responsibility of the publishing server to declare the URLs used for content

creation, retrieval and updates – all that AtomPub does is indicate which HTTP verbs are used to perform these actions.

Having said that, there are some loose conventions – retrieving content for a given resource feed is usually done via the form /*feedName*/content/, while updating or editing that content is done via the form /*resourcesName*/edit/, where /*resourcesName* reflects the name of the resource in question, but beyond this there's no explicit requirements.[8]

This lack may seem a little odd at first – how can you get the notation for a specific XML record, for instance? Again, however keep in mind that it is possible through the services interfaces (which are usually retrieved by passing the "introspection" URL that will have to be supplied with the given AtomPub server) to retrieve the applicable collection pointers, each of which includes the name of the URL for that particular collection. Once you

[8] The x2o server covered in the Details section covers a formal implementation of an AtomPub data server.

have the appropriate collection, you can retrieve at least some of the collection as a feed, which will in turn show how each entry is configured. This means that at any point you either know what the URL is already, or it can be inferred from the collection.

For instance, the coffeeshop's drink collection might be available as:
`http://www.mycoffeeshop.com/atom/coffeeshop/drink/content`

and would have it's edit "post address" as:

`http://www.mycoffeeshop.com/atom/coffeeshop/drinks/edit`

where http://www.mycoffeeshop.com/atom/ represents the atom server, `coffeeshop` is one feed, `/coffeeshop/drinks` is a subfeed (a feed that's defined as the child of another feed), and `content` and `edit` indicate the feed modality – how you interact with the feed.

Because these URLs can get to be long in practice, it's usual to drop the server in discussions. Thus,

`http://www.mycoffeeshop.com/atom/coffeeshop/drinks/content/`

is usually referred to simply as:

`/atom/coffeeshop/drinks/content/`

What's more, because what is more important to Atom is the HTTP method being used, it's typical to refer to add the method after the truncated URL. For instance, to post a new entry to an existing feed, you'd refer to it as:

/atom/coffeeshop/drinks/edit/ http 1.1/POST

This doesn't mean that all of this is the URL. Rather, it indicates that your atom client should send an atom entry to the URL:

```
http://www.mycoffeeshop.com/atom/coffeeshop/drinks/edit/
```

using the HTTP POST protocol.

Similarly,

```
/coffeeshop/drinks/edit/?myLatte
```

would be used to refer to the individual drink, though the exact mechanism for referencing individual entries will vary depending upon implementation.

In all cases, the success or failure of an AtomPub operation is indicated using the same HTTP response codes that are used with normal web pages. Thus, in the event of an http GET call, the response code would be 200 if the call was successful, 300 if the call performed a redirect on the server but was still successful, 400 if the resource couldn't be found, or 500 if an error occurred on the server.

The server code can be especially useful in those cases where an operation isn't expected to return a response, such as a PUT operation – this would typically return a 204 response code indicating that the action didn't return a response was nonetheless successful.

Additionally, most AtomPub servers will also return a header message explaining what happened in more detail, and most will also generate XML fault messages (though this is vendor specific) that can be parsed and used as appropriate.

AtomPub explicitly only defines operations for entries, but most AtomPub implementations can also perform the same type of operations for feeds. For instance, the eXist Atom implementation will let you create a new feed

container by POSTing a “template feed” XML to the URL to be created. This means that its possible for an AtomPub client to actually create a new resource feed as well as update (and delete it).

The Addressable Query

There’s been a bit of sleight of hand in the discussion up until now, specifically revolving around the question “what exactly is the difference between document and data?” This particular dichotomy has created a huge gulf between, on one hand, the world of the relational database management system (RDBMS) and, on the other hand, the world of XML and content management systems (CMS).

Increasingly, however, this distinction is disappearing. While this is happening, to a certain extent, due to the rise of XML as a way of encoding data, it’s also occurring because of two other aspects of data systems – *addressabilit*y and *queryability*.

In the syndication publishing model, one of the key characteristics of each resource as its pushed onto the stack is that the resource can be referenced via an address, albeit one that is assigned not by the user but by the publication server. When you request a feed for a given resource (assuming that paging is not in effect), in essence what you are doing is getting the addresses for every resource in the collection. Retrieving the representation from a given address will always correlate to one and only one “document”, with the additional significance that the key for that document is contained not in the document resource itself but in the metadata.

This last point is important – it means that if you have a relational table in a database, then it is always possible to identify which row in the table the given entry corresponds to (along with the analogous identification for an XML document in a collection of documents).

While people work with records in traditional relational database management systems (RDBMs), in reality, there is nothing in most RDBMs that correspond directly to the notion of a record. Instead, a set of records is generally the result of a SQL SELECT operation on a set of tables (again a collection).

As the Internet becomes more pervasive, a significant revolution is taking place within the data space. As discussed above you can serialize database records as XML data feeds (either with just raw data or with syndication related metadata). Database records typically are contextual – they are dependent upon the query that generated them, so normally, such records do not have globally identifiable identifiers.

However, suppose that you were to name the queries, and turn them into publishable data stores. Keep in mind one of the central precepts discussed earlier – a resource does not necessarily have to be a static object, but can instead be generated from other resources. Such a named query can then be treated as a resource in its own right that can be serialized as a syndication feed.

One question that emerges, though, with this approach is how to uniquely identify individual record resources from such a query. In a document-oriented approach, each document exists as a globally unique entity. For instance, suppose that you wanted to query all articles where the author's last name begins with the letter P. In a typical XML feed publishing approach, each document "record" has a distinctive metadata block including a globally unique identifier (such as a GUID, such as `1f92a592-38a9-45a7-b490-3bac43307eb7`).

In the case of a relational database, such identifiers cannot be generated at random, since the next time that the query is made the identifiers will

almost certainly be very different. Instead, it is likely that the system will have to create a hash function that will combine the query parameters (such as “author starts with P”) with locally identifiable characteristics within the record itself in order to create this key. Either way, though, this establishes a way for even “virtual” resources to be referenced from a URL.

This relationship between queries and resources is not accidental. A query is the most powerful means you have to create a subset of a given set of resources, of course, but it is also one of the means by which you construct virtual resources, and as such it is also one of the key mechanisms to make such virtual resources addressable on the web.

With both relational and xml based models, what is often of most interest is not the totality of all resources in a resource collection, but only those resources that satisfy a given query. For instance, a traditional feed of a card catalog might return simply a collection of all books that have been added to the publishing system. On the other hand, a queryable feed would let you select only those books (records) for which the author’s name starts with M or N, the books are science fiction, and the books feature female central protagonists.

Queryable feeds already feature fairly prominently in many larger web search engines (most of Google’s properties, YouTube, Flickr, and so forth feature queryable feeds), though in general the results that they return are links to web pages holding resources rather than to a specific XML representation of those resources. In essence, such feeds work by passing query parameters for a specifically defined (typically indexed text) query against the stored documents in a URL, then the server returns a news feed with only those items that have the specific search terms, given either in terms of a formal relevance ranking or by data published.

The next stage in the evolution of data services is to generalize queryable feeds to handle general XML content, rather than just "traditional" document content. To do that, however, you also need to create a more generalized data query language, one that can work well with data abstracted as XML as well as that can be integrated with the existing relational data model. While there are a number of proprietary candidates that companies would like to see as that data model, there is currently only one open standard for performing data query in that space – the *W3C XML Query Language*, otherwise known as *XQuery*.

The design philosophy in creating XQuery was to move beyond simply attempting to build a query language for XML and instead to attempt to unify the relational data model that begin with SQL twenty years ago with an awareness of distributed data streams, object-oriented programming, the growth of REST-based systems and a burgeoning awareness about the power of collections as an organizing principle for the web. For this reason, despite the obvious emphasis on XML in the name, XQuery has been designed to be a unifying data query language moving forward.

As a language, XQuery looks something like what you would get by mixing SQL, XPath, stored procedures and XML namespaces, and is broken down as follows:

XPath 2.0 Core. XQuery acts as a functional wrapper around the revised XPath 2.0 specification, which actually performs the bulk of the actual "selection" processing .

XPath 2.0 Function Set. XQuery also utilizes the considerably enriched function set that XPath 2.0 also exposes to allow for regular expression processing, date handling, document and text importation, advanced string and sequence operations, and similar capabilities.

FLOWR. XPath can select, but it can't process. The keywords For, Let, Order by,Where and Return are used to filter, sort, and shape nodes and collections for output.

Data Type Awareness. Unlike XPath 1.0, both XPath 2.0 and XQuery 1.0 are data-type aware, using the XSD atomic types to properly process given properties.
Function Module Extensions. Arguable one of the most significant aspects of XQuery, function model extensions make it possible to both define internal sets (or modules) of related XQuery functions that can be called like stored procedures as well as make it possible to extend the language with extensions written in Java, C# or whatever else the host language is.

XQuery can work in one of two basic ways, both of which are consistent with AtomPub. The first assumes that you have an intrinsic stored atom feed (a collection of entries) that are used as the source for the relevant query. In this particular case, the XQuery will generally filter specific entries out and then deposit what's left within the envelope of a second feed element. This approach is usually the most efficient, as it means that you can filter within the database directly then only serialize the output to an XML string once you have the resulting entries.

The second approach involves retrieving secondary collections, either from within an XML database or from external data feeds (or, in the case of most XQuery databases, from . This is often useful when you are attempting to perform look-ups – there a given field in one data feed acts as a key to retrieving other content that may not necessarily be stored within the resource itself. One benefit of this approach is that it makes it much easier to do localization and internationalization; as such look-up tables may contain different language versions of the same underlying content.

Either way, the upshot of this is that XQuery scripts can be written that make it possible to process data feeds and retrieve only those entries which satisfy a given criteria. Additionally, another facet that is making its way into a number of XQuery systems (most especially the XML database systems) is the introduction of function extensions which provides to the

user the ability to communicate with the web server session objects (request, response, session, etc.) as well as the back end data store.

These sets of extension functions in particular (and the extension mechanism in general) open up a very intriguing possibility for web application development. *By incorporating server capabilities into XQuery extensions, such systems can effectively end up replacing other server languages, such as PHP, ASP.NET, Ruby on Rails and so on*.

One consequence of this is the fact that when developing web applications in XQuery, the developer never needs to leave the context of XQuery in order to perform most web operations. Both structured data (XML) and binary resources (images, application files and so forth) can be retrieved and stored in a variable from the server and can be saved into the database all within the query itself. XSLT transformations can be invoked on incoming XML to create either output or input for another XQuery operation. SQL operations can be performed (both for query and updates), complex algorithms can be run (such as determining whether a geospatial point is in a given region), and most importantly, any data resource can be accessed as either a document or collection and the contents retrieved using XPath.

Given that a significant percentage of developing web applications involves creating plumbing between different data sources and processes, by establishing a single abstract data context, RESTful XQuery has the potential to significantly reduce the complexity of web application development by itself, and of course when combined with other ROA technologies can propagate that reduction in complexity all up and down the line.

Search and Discovery at the Edge of the Web

We've become spoiled by the search engine. Who would have imagined that you could encapsulate the ability to search the web into two simple controls – a text box and a button?

Unfortunately, this particular paradigm is beginning to run into problems. While there are no doubt countless optimizations, Google and other search engines work essentially by creating highly condensed versions of the web on hundreds of thousands of servers. As the web becomes increasingly frothy and marginally connected (think of the millions, perhaps now even billions of mobile devices that have only semi-persistent presences on the web) the strategy of indexing becomes increasingly error prone and inaccurate, not to mention placing significant strains upon the power generation of cities such as San Francisco, London, Tokyo, Singapore or Hyderabad.

What's more, staying in synch with changes to that worldwide resource database becomes an increasingly challenging proposition. Indexing machines can only be added at a linear rate of progression, while the number of web servers and data servers are increasing geometrically. This fact, known to the engineers working at the headquarters of most of the larger search engines world-wide, has caused many a sleepless night for those poor engineers.

At some point (and there are anecdotal indications that this time is close), it no longer becomes possible to index the web. As that time approaches, one of the approaches that most search engine organizations are taking is to focus not on the documents on the web but increasingly upon the metadata that is contained within syndication feeds, particularly the categorization metadata, and upon discovery mechanisms that are emerging again from the blogosphere.

The WS-* stack has a discovery mechanism called UDDI that was specifically designed as a "yellow-pages" approach to finding the location of specific services. One of the frequent charges against UDDI is the fact that its design tends to encourage the development of centralized repositories that work at odds with the increasing need for distributed discovery.

The ROA discovery process that seems to be evolving is somewhat more layered. One approach to discovery is the use of XML-based "outline" formats that started out as ways of syndicating "blog-rolls", lists of blogs a given site may recommend as worthwhile reads. Such lists include not only link-rich connections to specific blog feeds but can also contain a certain level of categorical metadata which indicates the rationale for the link being listed.

The current mindshare leader in this space is the Outline Processor Markup Language (**OPML**) language. This XML-like language can be expressed either in XML or in a more HTML-like format, and was, like many first generation syndication formats a language that was originated by and heavily pushed by Dave Winer. Problematically, it doesn't handle namespace extensions or categorization all that well, instead using a fairly unconventional approach to extending the language that is not consistent with many other XML languages.

The HTML 5.0/WHAT WG group have proposed an alternative discovery method that's specifically driven by microformats working in conjunction with the HTML element. In this case, the link element includes a `rel` attribute with a value of `"feed"`. While this has the advantage of being able to take advantage of the growing development work in microformats and "semantic web lite", its principle disadvantage is that it makes the assumption that HTML will be the only content that has a use for discovery.

Of course, such a discover process could also be built into an Atom feed directly, specifically within the feed links section. Typically, most Atom feeds provide a direct link to themselves of the form:

```
<link
href="http://www.myfeed.com/atom/content/blogs/kurt"
type="application/atom+xml"
rel="alternate"/>
```

where `/atom/content` indicates that the feed returned should use the defined atom services (in this case the `/atom/content` service) and `/blogs/kurt` indicates that the `/kurt` feed within the `/blogs` feed should be serialized (this ability to create feeds within feeds is implementation dependent, but a powerful capability for organization nonetheless that will be covered in greater detail in the details section). The `rel="alternate"` attribute (where rel is shorthand for the relationship of the link) is the significant piece of information here, however, in that it identifies the location of the resource in question for news readers and news services.

Query and Services

A RESTful Approach to Services

A language limited solely to verbs is active but confusing (everything becomes a command). Similarly a language limited solely to nouns is well-defined but dull. Just as SOA works upon the assumption that the invoked verbs take either parametric content from a query or work against some internal data source, so too do ROA systems integrate services, with the primary difference being that a service is generally something that can be expressed within a URL and (almost always) works upon a stated resource feed.

One caveat that should be covered before continuing is that ROA services have both a conceptual view and an implementation view for URLs. The reason for this is simple – RESTful systems should in general respect the internal conventions and capabilities that different servers utilize for specifying formats, making it difficult to create one-size-fits-all URLs for every different potential platform.

Thus, one system may refer to an Atom feed by:

```
http://www.resources.com/atom/content/blogs/kurt
```

while another may use the notation:

```
http://www.resources.com/atom/content.xq?blogs/kurt
```

and yet another as:

```
http://www.resources.com/?serv=atom/content;res=blogs/kurt
```

Because of this, AtomPub jump starts the process of identifying, once you do find the site, where the resources (and their associated services) are located. An AtomPub services document provides both a primary feed URL for each feed defined on the system and will indicate which particular services are publicly exposed, and in general the location of this services document is the one piece of the puzzle that has to be user supplied (although most blogging clients, the earliest users of AtomPub, can make intelligent guesses based upon knowledge of the particular server).

The services document in turn identifies the specific URIs for each collection of resources. In a blogging-oriented AtomPub server, it's generally sufficient just to have one such URI per resource, because the verbs that are doing the heavy lifting are the HTTP verbs defined earlier (GET, PUT, POST, DELETE and HEAD). Thus, posting a new entry to a typical AtomPub collection would use the same URI as that used to retrieve the Atom-pub feed in the first place, but would use the POST verb in the HTTP header rather than GET.

However, that doesn't preclude "super-classing" the operations with different services. For instance, for security reasons, it may be generally preferable to split the ability to edit content within a feed store from the ability to view it. Thus, you may actually want to define two distinct services /content and /edit that can subdivide the capabilities between them, such that /content would only accept GET or HEAD, while /edit would be able to handle all of the primary HTTP verbs. For example,

```
http://www.resources.com/atom/content/blogs/kurt    HTTP 1.1/GET
```

might be a permissible service, while

```
http://www.resources.com/atom/content/blogs/kurt    HTTP 1.1/POST
```

might not – you might need to use

```
https://www.resources.com/atom/edit/blogs/kurt    HTTP 1.1/POST
```

instead, which insures that you're sending content across a secured channel.

However, it is worth emphasizing here that these are still just URLs – specific addresses that provide different interpretations of the five HTTP verbs. Similarly, working with individual records usually necessitates using URLs that will be implementation dependent. For instance, if your AtomPub server assigned the relevant internal IDs, then retrieving a feed containing JUST that entry might be rendered as:

```
/atom/content/blogs/kurt/?id=0e8976426839
```

One aspect of presenting these multiple "services" is that different services could render different presentations of the same feed entries in different ways. For instance,

```
/atom/edit/blogs/kurt/?id=0e8976426839  HTTP 1.1/GET
```

might very well return the given entry not as an atom feed but as an XForms document that will let you edit the underlying XML of the feed, and that would then POST the XML as an entry to the AtomPub server automatically to the same address, providing a way of automating the editing process directly from the web.

In a similar fashion:

```
/atom/json/blogs/kurt  HTTP 1.1/GET
```

may return an Atom feed not in XML notation but using the JavaScript Object Notation instead, with POST and PUT acting to take in JSON feeds in

much the same manner. Internally, the server will reconstruct the XML and persist it into the data store (or will serialize the output from XML to JSON) but this behavior is opaque to the end-user – they just see it as a server that will handle their respective feeds.

Indeed, this can even be carried to its logical conclusion of saying that you could even conceivably wrap your XML content within a SOAP wrapper and send that to:

```
/atom/soap/blogs/kurt  HTTP 1.1/POST
```

This last possibility should be examined very carefully. A SOAP message is, at its core, a serialized command. A REST based architectural system assumes no semantics – it doesn't in fact know what to do with those commands, nor should it.

However, one way of thinking about a feed is that it is a persistent queue. If you post a set of SOAP commands to such a queue, there is nothing that says a separate process, independent of and asynchronous to the publishing process, couldn't in turn retrieve some or all of the SOAP "entries" that have not yet been "published" and run a marshalling process on them before switching them from draft to published mode (something that *is* in the AtomPub specification).

This means that it is possible to use AtomPub to build a very compelling "bridge" to a SOA-based architecture, one that has the additional benefit of automatically archiving each transaction. At the same time, regardless of the input service, the resources themselves retain their internal integrity because people are only dealing with them through these services facades.

Query Redux

Given the deployment of such “named” services in this manner, it’s worth readdressing where XQuery fits into this mix. In point of fact, if you have a REST aware XQuery system, it may very well be that the code to back each of these URLs is essentially an XQuery call.

XQuery is designed to work well with collections, which the entries making up a data feed most certainly are. This means that certain basic operations (ordering by latest feed, paging, and similar activities) can readily be done by assuming that you have a data feed then use the XQuery to filter only those elements that satisfy a given criterion. Similarly you can use XQuery to parse feeds into or out of specific formats (such as the aforementioned JSON and SOAP AtomPub URLs).

Queries can take two forms in a restful environment. The first is, as mentioned, associating a given query with a base URL that works against a named feed. Thus, if your data-set consisted of something like emergency response calls in descending order, then you could associate a URL service called something like `/atom/responseloc` with an XQuery script (call it `response.xq`) that would take as parameters a rectangle with city grid coordinates in the form upper left-lower right and would and would then return a feed listing all of the responses that occurred in that area:

```
/atom/responseloc/responses/eastside?region=45.332,38.212,45.312,38.334
```

Internally, there would be a mapping XML structure that created the association between `/atom/responseloc` and `response.xq`:

```
<service name=”responseloc”>
    <method name=”GET” action=”/queries/response.xq”/>
</service>
```

and the function would then parse out the query string to get the coordinates and apply them to the existing feed (this assumes that each

entry has encoded the location of the call using geoRSS in the metadata section of the entry):

```
declare namespace atom="http://www.w3.org/2005/Atom";
declare namespace geo="http://www.w3.org/2003/01/geo/wgs84_pos#";

let $base-feed := /atom:feed
let $region := request:get-parameter("region")
let $rect := tokenize($region,",")
let $feed := <atom:feed>
   <atom:title>Events for ({$rect[1]},{$rect[2]}) to
({$rect[3]},{$rect[4]}</atom:title>)
   <!-- additional feed metadata -->
   {for $entry in /atom:feed/atom:entry where
$entry/geo:lat ge $rect[1] and
$entry/geo:lat le $rect[3] and
$entry/geo:long ge $rect[2] and
$entry/geo:long le $rect[4]
order by atom:updated descending
return $entry}
</atom:feed>
return $feed
```

This particular form of query is useful for commonly occurring operations. Significantly, there are a number of existing web service architectures, such as the OpenGIS platform, that has encoded exactly these types of queries into their systems, using a rather complex and cumbersome services stack to do so that could benefit dramatically from the rethinking of their architecture in terms of more purely ROA terms.

However, it is the second approach to querying that opens up possibilities. An XQuery is itself a "document", albeit not necessarily an XML one. Given that, it should be possible to post as an atom entry an XML document that consists of a query name and title, a mechanism for describing expected parameters, and the query itself. Once posted, the query is essentially assigned as a resource within the database with a name (and/or extension

or location) that would trigger its invocation as an XQuery call. For instance, a script similar to that above might be encoded in an entry as follows:

```
<atom:entry xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:x2o="http://www.metaphoricalweb.org/xmlns/x2o">
      <atom:title>Events In Region</atom:title>
      <x2o:alias>events-in-region</x2o:alias>
      <atom:category term="xquery"/>
      <atom:abstract>This feed returns all events that have occurred
with a rectangle of the form rect=lat1,long1,lat2,long2 (left to
right)</atom:astract>
      <atom:content type="application/xquery"><![CDATA[
declare namespace atom="http://www.w3.org/2005/Atom";
declare namespace geo="http://www.w3.org/2003/01/geo/wgs84_pos#";

let $base-feed := /atom:feed
let $region := request:get-parameter("region")
let $rect := tokenize($region,",")
let $feed := <atom:feed>
   <atom:title>Events for ({$rect[1]},{$rect[2]}) to
({$rect[3]},{$rect[4]}</atom:title>)
   <!-- additional feed metadata -->
   {for $entry in /atom:feed/atom:entry where
$entry/geo:lat ge $rect[1] and
$entry/geo:lat le $rect[3] and
$entry/geo:long ge $rect[2] and
$entry/geo:long le $rect[4]
order by atom:updated descending
return $entry}
</atom:feed>
return $feed
]]>
    </atom:content>
</atom:entry>
```

This feed would then be posted to

```
/atom/system/queries
```

which would perform the same basic operations as something like `/edit`, but would also make the resource in question invocable (the specific details of which are beyond the scope of this paper).

Once the query resource has been saved (and given an id of `events-in-region`), you can use it on other resource feeds:

```
/atom/query/responses/eastside/?q=events-in-
region;rect=45.332,38.212,45.312,38.334
```

In other words, using a simple RESTful interface, you give to (presumably privileged) users the ability to create rich and complex queries that can be persisted on the server directly from a standard web page interface, an XForms document, or some other UI. Moreover, because such queries run within the context of specific user permissions, you can also insure that only those users that are especially trusted can create queries that have the ability to modify the database itself.

As data becomes increasingly complex, the domain experts rather than the system administrators will become the most familiar with the underlying structure of the data models, and giving into their hands the ability to query the data (even to the extent of generating rich reports or virtual resources) moves data management issues increasingly in to the hands of the data users.

From XQuery to XUpdate

A challenge that XQuery has faced for some time has been that it was primary a read-only language – it was if you have taken SQL and implemented only the SELECT portion of the language. Obviously, in order to use XQuery to build a publishing system, the language needs to have a

mechanism for updating the back-end data store, so this lack seems somewhat peculiar.

In point of fact, the need for updating XML databases and XML-abstracted back-end systems has been evident for some time, and as a general rule there are no XQuery implementations currently in use that do not provide some way to update content through XQuery. Indeed, many use the XQuery extension mechanism itself to provide this functionality, although this is also combined in specific cases with a means of interacting with relational databases using SQL.

Despite this, moving forward an update mechanism is needed. To that end, the W3C XQuery working group has been putting together its own standard for performing updates – the XQuery Update Facility (alternatively referred to as **XUpdate** and **XUF**, though XUpdate is marginally more euphonious). XUpdate has been in development in parallel with the XQuery process, and should become a formal recommendation by Summer 2008 (it's in Candidate Recommendation status at the time of this writing).

The XQuery Update Facility is broken down into three distinct parts. The first establishes a set of six operations that can be used to modify a given XML structure:

Insert – for placing new nodes or content into an existing structure
Delete – for removing nodes from a structure
Replace – which can either replace one set of nodes with another or replace a value that a given node has with another value.
Rename – for changing the names of element or attributes
Transform – for creating modified copies of nodes

where a node in this case refers to an element or attribute within an XML structure.

The second part of the specification makes some changes to the current XQuery syntax mechanism to support the operations at an implementation standpoint for optimization while still maintaining backwards compatibility with existing XQuery implementations.

The final part provides a set of functions specifically oriented towards expanding the XPath 2.0 functions to better handle the operations specified above. This includes such things as invoking a revalidation of an existing XML structure after a change has been made, putting new XML resources into the database, changing the associated type of a given node, and combining multiple update operations so they can all be batched and hence made more efficient.

Most of the XML Database vendors have been tracking the changes to this specification closely (indeed, many are the ones writing the specification), and will be incorporating these changes into their XQuery products within months of the formal release of the XUpdate specification.

Shaping Data with XForms and MVC

Once you get beyond the assumption that the only possible content that an Atom feed can carry or an AtomPub server can process is blog content, the model for working with any object that can be represented in XML begins to open up fairly dramatically. However, one facet of the ROA architecture consequently becomes both more critical, and that is to a certain extent still under-served – the editing layer.

The days when you could express a data model as a handful of properties has long since passed. The typical business taxonomy contains hundreds (and in many cases, thousands) of properties that are related to one another

in dizzyingly complex ways with constraints that are both convoluted and dynamic. Given that, working with name/value pairs being passed on a URL's query string seems … quaint … at best, yet this is still how applications are being constructed even at Fortune 500 companies.

An approach that is beginning to gain steam in the industry is model based development. The idea here is surprisingly simple – make the model the centerpiece of your development efforts, not the processes that act on that model, then set up an architecture on the various nodes in your network, whether user-facing or machine-facing, that can essentially manipulate instances of the data model.

Resource oriented development is a direct reflection of this approach, as a resource can be thought of as being an instance of a data model for a given object along with a wrapper of publishing metadata associated with that object. On the server side, this is seen on the focus on collections of resources with associated URIs that describe different manifestations of these resources. On the client side, what is needed instead is a way of articulating the data model in a way that keeps the resource more or less intact.

The Model/View/Controller design pattern, otherwise known as **MVC**, has been around since the advent of the SmallTalk language in the 1970s. In an MVC application, the developer establishes an application data model then writes a controller that communicates with a user interface view. Typically this means that each user interface component (such as a text box) is either directly or indirectly bound to some aspect or property within the data model – when the model changes, if that component falls within the change scope then it will reflect the change. Similarly, if the component is an editing control, then changing the value of the control will cause a corresponding change via the controller into the data model itself.

In 1999, with the XML specification completed, attention turned in the W3C to reworking HTML as an XML-based language, rather than as an XML-like one. While most elements made the transition over without much real effort, the forms-based components (which had, after all, been a comparative late comer to the HTML specification) made people ask whether there was some way that web forms could be designed to pass XML structures rather than name-value pairs.

In 2000, a separate XForms activity spun off the W3C working group, tasked with aligning XML with XHTML, and the mandate quickly jumped from simply providing a way to let the controls hold independent values to creating a binding architecture that implemented a first class MVC model in XML. The first XForms 1.0 Recommendation was published in 2002.

XForms success in the marketplace has been what could charitably be called *mixed*. Implementing XForms engines is non-trivial, especially since the company with the largest browser share in the market had chosen to ignore XForms in favor of proprietary technology that existed for the most part outside of the browser. Early vendors into the space found it hard to explain the rationale for an XForms engine, given that it, like most model-driven technologies, necessitated a cleaner understanding of what the underlying data model looked like than was typically the case for web applications.

Moreover, most web pages were built with an orientation towards low level consumer-facing applications; while deploying larger scale applications to the browser has long been a dream for enterprise IT managers, the complexities of the models involved made developing and deploying such systems difficult, error prone and fragile in the face of evolving business conditions.

Developments in 2007 and 2008 suggest that XForms may have a considerably brighter future. The XForms 1.1 working draft will become

finalized in the summer of 2008, fixing a number of long-standing issues with the 1.0 specification that have limited deployment. Server-based XForms projects have become quite mature and stable, and XForms is appearing implemented in mobile chipsets, as well as within a new generation of mid-tier layers tied into Adobe Flex and Microsoft Silverlight, though not necessarily components produced by those companies directly. In other words, XForms is now reaching a level of penetration where it becomes feasible to deploy across systems.

For all that, however, perhaps the brightest future that XForms has is in conjunction with ROA based systems. XForms can consume both Atom feeds and their associated content resources, and can send Atom feeds (and atom entries) back to the server as distinct, discrete entities. XForms implementations. XForms is also well suited to the types of larger schemas that are increasingly the norm for ROA based applications, being able to render various parts of a given model as relevant or irrelevant at any given time, communicating effectively with server systems using the core HTTP verbs, and working especially well with the paging and querying aspects that are a major part of AtomPub's attraction as a data publishing API.

As mentioned in the previous section, XForms "pages" can be presented as one service view by AtomPub. Moreover, the same kind of user-centric publishing mechanism that XQuery can use (the second type of queries cover in the previous section) can also be utilized for XForms, such that a given XForm "interface" can be published to the AtomPub server as its own resource and then be deployed as a named editor that can be used to either create new resources from a base template or modify an existing resource.

One final aspect about XForms makes them especially attractive as resource oriented architectures become more prominent. It is possible using schemas and related modeling tools as sources to generate (via one or more XSLT

transforms) a “preferred” Xforms document that can capture much if not all of an underlying XML data model instance.

This can often be used to create just-in-time editors that cut down dramatically on the amount of user interface development necessary to build web-based applications, making such applications far more attractive in situations where the cost of developing such applications normally significantly outweighs the benefits to automating these systems, especially when coupled with Atom(Pub) and XQuery based systems.

Indeed, the combination of XQuery, REST (Atom) and XForms is so potent that an acronym, ascribed first to Dan McCreary of McCreary and Associates, is now gaining currency in both the XQuery and XForms community – XRX. XRX systems are perhaps archetypal resource oriented architectures, reflecting an awareness that while there are certainly other related technologies and similar stacks, it is this combination of XML data abstraction layer, XML data transport layer and XML data presentation layer that represents a major change in the way that applications will be built moving forward.

Conclusion

Resource oriented architectures, built around the concept that a large conceptual space (such as the web) can be broken down into collections of resources, manipulated via atomic, data-publishing primitive operations, queried via a data abstraction mechanism and mediated via syndicated feeds, is one of the most significant shifts in the underlying information models of our time. It provides a counterweight to the burgeoning SOA movement, one which is focused increasingly on resources rather than

actions and on decoupled, low-level publishing operation systems rather than on tightly bundled coupled ones.