Articles

Resource-Oriented Architecture: The Rest of REST

Author: Brian Sletten

Series Introduction

Think for a moment, if you can, back to a time before the Web. Imagine trying to explain the impending changes to your hapless contemporaries. It is likely they would simply not be able to fathom the impacts that the Web's emergence would have on nearly every aspect of their lives. In retrospect, it feels like a tsunami caught us off-guard and forever altered the landscape around us. The reality is more pedestrian, however. It was a deliberate series of technical choices that built upon each other that yielded the results we have experienced.

Now, pause and reflect upon the idea that you are probably in a similar position to those incredulous pre-Web types you were just trying to enlighten. Unless you have been paying close attention, you are about to be caught off-guard again as it feels like a new wave crashes upon our economic, social, technological and organizational landscapes. While the resulting changes will feel like they occur overnight, the reality is that they have been in the works for years and are just now producing tangible results. This new wave is about a Web that has evolved beyond documents into Webs of Data, both personal and private. We will no longer focus on information containers, but on information itself and how it is connected.

This wave has been in the works for years and is again being driven by the deliberate adoption of specific choices and technologies. These choices are combining to solve the problems caused by the inexorable march of technological change, business flux, new and varied data sources and the ubiquitous, expensive and failure-prone efforts that have cost millions and delivered insufficient value. Web Services and Service-Oriented Architectures (SOA) were supposed to be part of the answer, but the elegance of their visions have been forever stained by the inelegance of their technical solutions.

The beauty is that we are not starting from scratch. We are building upon the technology we have in place to grow these data webs organically. We can wrap our databases, libraries, services and other content sources with a new set of abstractions that will help us off the treadmill we have been on. We are integrating the public Web of Data with our own, privately held data. The incremental adoption of these technologies is yielding new capabilities that will, in turn, unlock further capabilities.

This is the first article in a new series to highlight the evolution of information-oriented systems that got us to where we are and provide a roadmap to where we are going. Despite what it may seem on the surface, these choices are neither ad hoc nor esoteric, but rather foundational decisions based on a long tradition of academia and applied engineering.

We will start by revisiting the REpresentational State Transfer (REST) architectural style. Oft quoted and even more often misunderstood, this manner of building networked software systems allows us to merge our documents, data and information-oriented services into a rich, logical ecosystem of named resources. From there, we will introduce the vision of the Semantic Web and walk through its core technologies represented by a flexible and extensible data model and the ability to query it. We will see how to incorporate relational data, content from documents, spreadsheets, RSS feeds, etc. into a rich web of reusable content.

After we present the basics, we will walk through a variety of successful efforts building on these technologies and then return to reclaiming the vision promised to us by proponents of Web Services technologies. We will describe a process where we can achieve something of a Unified Theory of Information Systems; one that not only handles, but embraces the kind of technical and social change that has been painful and intractable to manage in the past.

There has been too much hype surrounding the Semantic Web, but there have also been a steady stream of quiet successes. This series will be a pragmatic guide into both new and familiar territory. We will connect the technologies in deeper ways than perhaps you have seen before. We will highlight events and actions by companies, government organizations and standards bodies that indicate that this is happening and it will change everything. We will show how a very large difference in your system implementation can often be made through subtle shifts in perspective and adoption of standards that are designed to facilitate change.

The first step, is to embrace a common naming scheme for all aspects of our infrastructure. A Service-Only Architecture usually ignores the data that flows through it. At the end of the day, our organizations care about information first and foremost. REST and the Web Architecture puts this priority up front and lays the foundation for the remainder of our discussion.

The Rest of REST

It has become fashionable to talk about the REpresentational State Transfer (REST) as something of a weapon in the War On Complexity. The enemies in this war, according to some, are SOAP and the Web Services technology stack that surrounds it. This Us vs Them rhetoric brings passion to the table, but rarely meaningful dialogue so people remain confused as to the underlying message and why it is important. The goal is not to replace SOAP; the goal is to build better systems.

REST is not even a direct replacement for SOAP. It is not some kind of technology of convenience; a simple solution for invoking Web Services through URLs. The management of information resources is not the same thing as invoking arbitrary behavior. This confusion leads people to build "RESTful" solutions that are neither RESTful, nor good solutions.

REST derives its benefits as much from its restrictions as it does its resultant flexibility. If you read Dr. Roy Fielding's thesis (which you are encouraged to do), you will learn that the intent was to describe how the combination of specific architectural constraints yields a set of properties that we find desirable in networked software systems. The adoption of a uniform interface, the infamous Uniform Resource Locator (URL), contributes to the definition of REST, but is insufficient to define it. Likewise, interfaces that simply expose arbitrary services via URLs will not yield the same benefits we have seen so successfully in the explosion of the Web. It takes a richer series of interactions and system partitioning to get the full results.

Most people understand that REST involves requesting and supplying application state of information resources through URLs via a small number of verbs. You retrieve information by issuing GET requests to URLs, you create or update via POST and PUT, and remove information via DELETE requests.

This summary is not incorrect, but it leaves too much out. The omissions yield degrees of freedom that unfortunately often allow people to make the wrong decisions. In this gap, people create URLs out of verbs which eliminates the benefit of having names for "things". They think REST is just about CRUD operations. They create magical, unrelated URLs that you have to know up front how to parse, losing the discoverability of the hypertext engine. Perhaps most unforgivably, they create URLs tied solely to particular data formats, making premature decisions for clients about the shape of the information.

Understanding the full implications of REST will help you avoid these problems; it will help you to develop powerful, flexible and scalable systems. But it is also the beginning of a new understanding of information and how it is used. Upon this foundation of Web architecture, the application of the remaining technologies of the Semantic Web will yield unprecedented power in how we interact with each other as individuals, governments, organizations and beyond. This is why we begin with a deeper dive into the parts of REST that many people do not understand and therefore do not discuss. These topics include the implications of:

- URLs as identifiers
- Freedom of Form
- Logically-connected, Late-binding Systems
- Hypertext as the Engine of State Transfer (HATEOS)

URLs as Identifiers

We have already established that most people know about URLs and REST. It seems clear that they understand that a URL is used for invoking a service, but it is not clear that they get the larger sense of a URL as a name for information. Names are how we identify people, places, things and concepts. If we lack the ability to identify, we lack the ability to signify. Imagine Abbott and Costello's infamous "Who's on First?" skit on a daily basis. Having names gives us the ability to disambiguate and identify something we care about within a context. Having a name and a common context allows us to make

reference to named things out of that context.

The Uniform Resource Identifier (URI) is the parent scheme. It is a method for encoding other schemes depending on whether we want them to include resolution information or not. Librarians and other long-term data stewards like names that will not change. A Uniform Resource Name (URN) is a URI that has no location information in it; nothing but name is involved. The good news is that these names will never break. The bad news is that there is no resolution process for them. An example of a URN is an ISBN number for a book:

```
urn:isbn:0307346617
```

In order to find more information about this book, you would have to find a service that allows you to look up information based on the ISBN number.

If nothing about the context of our systems and information ever changed, we would probably always want to include resolution information in our resource names so we could resolve them. But anyone who has been handed a broken link knows we want longer-lived names for really important stuff. Looking at our history of using URLs, we have done some silly things when we created ones such as:

```
http://someserver.com/cgi-bin/foo/bar.pl
```

```
http://someserver.com/ActionServlet?blah=blah
```

```
http://someserver.com/foo/bar.php
```

The problem with these URLs is that the technology used to produce a result is irrelevant to the consumer of information. There is no good reason to create URLs like that. The focus should be on the information, not the technology. Implementation technologies change over time. If you abandon them, for instance, any system that has a link to the Perl, Servlet or PHP-based URL will break. We will address some infrastructure to solve this problem in future articles, for now, we will just try to make careful choices in the names we give our information resources.

Despite being fragile, the URL scheme does allow us to disambiguate information references in a global context.

```
http://company1.com/customer/123456
```

is distinct and distinguishable from

```
http://company2.com/customer/123456
```

in ways that a decontextualized identifier like '123456' is not.

To ground the concept into a larger information systems framework, you can think of a URL as a primary key that is not specific to a particular database. We can make references to an item via its URL in dozens of different databases, documents, applications, etc. and know that we are referring to the same thing because we have a unique name in a global context. We will use this property in future discussions to describe and connect RESTful systems to other content and metadata.

The next aspect of URLs that bears discussion is their universal applicability. We have a common naming scheme that allows us to identify:

- documents (reports, blogs, announcements)
- data (results, instance information, metadata)
- services (REST!)
- concepts (people, organizations, domain-specific terms)

We do not need to come up with a different mechanism to refer to each different category of things. A careful application of some specific guidelines allows us to blur the distinctions between these things which brings us to the last point for now about URLs. Not only are these names useful in order to refer to information we care about, but systems that receive these references can simply ask for them. The 'L' in URL (locator) gives us the capacity to resolve the thing, not knowing anything else about it. We can usually invoke the same basic operations on everything we can name. Issuing a GET request to a URL representing a document, some data, a service to produce that data or an abstract, non-network-addressable concept all work fundamentally the same way. For those things we have the permission to manipulate, we can also create, modify or delete them using similar means.

Freedom of Form

Our experience of the Web has been somewhat passive with respect to the shape of information. When we click on a link, we expect the content to come back in a particular form, usually HTML. That is fine for many types of information, but the architecture supports a much more conversational style allowing clients to request information in a preferred form.

To understand why this is useful, consider a company's sales report. It is easy to imagine this being useful to executives, sales people, other employees, clients and investors as an indication of how a company is performing. A possible name for such a report could include the year and the quarter in the URL:

```
http://company1.com/report/sales/2009/qtr/3
```

We might contrast this with a sales report for the month of March:

```
http://company1.com/report/sales/2009/month/3
```

Both are good, logical names that are unlikely to break over time. It is a compelling vision that people could simply type such a URL into a browser and get the information they seek rendered as HTML. The reports could be bookmarked, e-mailed, linked to, etc.; all the things we love about the Web.

The problem is that the information is locked into its rendered form (until we introduce technologies like GRDDL and RDFa later in this series!). We used to try to scrape content from pages, but gave up in disgust. As the page layout changes, our scripts break.

If you were a programmer for this company and wanted to get to the information directly, you might like to request it as XML. You could get back raw, structured data that you could validate against a schema. HTTP and REST make this trivial as long as the server knows how to respond. By passing in an "Accept: application/xml" header to your request, you could indicate a preference (or requirement) for XML. On success, you will get back a byte-stream with a MIME type indicating that your request has been honored. On failure, the server will indicate via a 406 Error that it cannot help you. In that case, you might want to contact the department responsible for this information and request they add the support you need; something they can do without breaking any existing clients. If you were a business analyst, you might think that XML has sharp points and can hurt you, so you might like to request it back as a spreadsheet, a format that is easily incorporated into your existing workflows, tools and processes.

The point is that the logical name for the report is easily converted into various forms at the point it is requested. It is equally easy to run systems that accept modifications back in the various forms. The client has no visibility into how the information is actually stored, they just know that it works for them. This freedom is wholly underused by people building RESTful systems. When they stand up a service and decide that they will only return XML, they miss the potential value REST has to an organization.

Because many developers are either unaware of content negotiation or find it difficult to test in a browser, they define different URLs for the different formats:

```
http://company1.com/report/sales/2009/qtr/3/report.html
```

```
http://company1.com/report/sales/2009/qtr/3/report.xml
```

```
http://company1.com/report/sales/2009/qtr/3/report.xls
```

This developer convenience becomes a limitation once you escape the confines of a particular use. In essence, we now have three information resources, not one that can be rendered in different forms. Not only does this fork the identity in the global context, it also prematurely commits other clients to a particular form. If you pass a reference to a URL as part of a workflow or orchestration you are robbing the upstream clients from the freedom to choose the form of the data.

There are several ways to test a proper RESTful service without using a browser, for example:

```
curl -H "Accept: application/xml" -O
http://company1.com/report/sales/2009/qtr/3
```

using the popular curl program. Any reasonable HTTP client will provide similar capabilities.

The benefits of supporting a rich ecosystem of negotiable data forms may not be immediately obvious, but once you wrap your head around it, you will see it as a linchpin toward long-lived, flexible systems that favor the client, not the developer.

Logically-Connected, Late-Binding Systems

Once you commit to good, logical names for your information resources, you will discover some additional benefits that fall out of these decisions. Named references can safely and efficiently be passed back as results without returning actual data. This has strong implications for large and sensitive data sets, but it also makes possible technical and architectural migration.

For the same reasons pointers are useful in languages like C and C++, URLs as references to data are more compact and efficient to hand off to potential consumers of information. Large data sets such as financial transactions, satellite imagery, etc. can be referenced in workflows without requiring all participants to suffer the burden of handling the large content volume.

Any orchestration that touches actual data must consider the security implications of passing it on to other systems. It quickly becomes untenable to provide perfect knowledge of who is allowed to do what at every step of a process. If a reference is passed from step to step, it is up to the information source to enforce access. Some steps may not require access to the sensitive information and could therefore be excluded from receiving it when they resolve the reference.

This means the late-binding resolution can factor in the full context of the request. A particular user accessing a resource from one application might have a business need to see sensitive information. The same person using a different application might not have a business justification to the same data. A RESTful service could inspect session tokens and the like to enforce this access policy declaratively. This level of specificity is required to prevent internal fraud, often the biggest risk in systems that deal with sensitive content. The details of such a system are going to be implementation-specific and are largely orthogonal to the process of naming and resolving logically-named content.

Dependency on a logical connection allows clients to be protected against implementation changes. When popular websites shift from one technology to another, they are usually successful at hiding these changes from their users. RESTful services do the same thing. This gives us the freedom to wrap legacy systems with logical interfaces and leave them in place until there is a business reason to invest in a new implementation. When that happens, clients can be protected from being affected.

In addition to mediating technology changes, RESTful systems allow you to embrace a variant of Postel's Law: Be Conservative in what you do; be Liberal in what you accept from others. You can maintain strict content validation of what you accept and return. However, if you have an existing client base that is providing you content in a given form, you are free to allow other clients to provide different forms, different schemas, etc. without affecting the existing clients. Systems that closely associate a contract with an endpoint tend not to have this freedom which makes them more brittle and quickly fragmented.

Hypertext As the Engine of State Transfer (HATEOS)

As systems come across references to information resources, many people think there needs to be some sort of description language to indicate what is possible or should be done with it. The reality is that a well-considered RESTful system usually does not require this concept. This is difficult for SOAP developers to accept, but it has to do with the constraints of the architectural style. Because we treat information resources as things to manipulate through a uniform interface (the URL!) and restrict our efforts to a small set of verbs, there really is no need to describe the service.

If you find yourself confused on this point, it is probably an architectural smell that you are conflating manipulating resources with invoking arbitrary behavior. The REST verbs provide the full set of operations to apply to an information resource. Certainly, you need to know what information is being returned so you know how to process it, but that is what MIME types are for. While it is usually preferable to reuse known types (application/xml, image/png, etc.), many developers do not realize that they can create their own application-specific data types if necessary.

In the larger arc of this article series, we will address the problems of finding and binding arbitrary resources using rich metadata. For now, we will simply keep in mind Roy's underscoring of the importance of "hypertext as the engine of state transfer" (obliquely referred to as "HATEOS" by RESTafarians). This is perhaps the most misunderstood portion of the thesis. To get its full implication, we need to revisit how the Web works.

You type a URL into the browser and it issues an HTTP GET request for that resource. Invariably, the server responds with a bytestream, a response code (usually 200 on success) and a MIME type indicating that the response is HTML. The browser decides it knows how to handle this type and parses the result into a document model of some sort. Within that model, it finds references to other resources: links, images, scripts, style sheets, etc. It treats each one differently, but it discovers them in the process of resolving the original resource. There is no service description; the browser, as a client, simply knows how to parse the result.

The same mechanism should be employed for RESTful services. The URLs themselves should not be "magical". A client should not be required to know how to parse a URL or have any special knowledge of what one level in the hierarchy means over another one. RESTful clients should retrieve a resource, investigate the returned MIME type and parse the result. As such, a client should know how to parse the returned type.

For example, a client might receive a reference to the main RESTful service for the reporting service we described above:

```
http://company1.com/report/
```

If requested from a browser, it could return an HTML document that has references to:

```
http://company1.com/report/sales
```

which the user could click through to find a list of years to browse. The point is that the browser has

no special knowledge of the URL structure, but it knows how to parse the result and present the content to the user in a way she can explore.

The same can be true of other MIME type responses. For example, requesting the 2009 quarterly reports as XML:

```
http://company1.com/reports/sales/2009/qtr
```

could yield:

```
<reports>
   <description>2009 Quarterly Reports</description>
   <report name="First Quarter"
src="http://company1.com/reports/sales/2009/qtr/1"/>
   <report name="Second Quarter"
src="http://company1.com/reports/sales/2009/qtr/2"/>

   <report name="Third Quarter"
src="http://company1.com/reports/sales/2009/qtr/3"/>
</reports>
```

You can think of the URL as a vector through an information space. Each level points you closer to the ultimate resource. Different paths can yield the same results. The client will have to know how to parse these results, but by giving the response an identifiable type, we can trigger the appropriate parser. The structure can be spidered by descending through the references, or presented to a user to browse through some kind of interface. A RESTful interface becomes a way for clients to ask for information based on what they know. They start from a known or discovered point and browse the information like you browse the Web.

This is what HATEOS refers to. The application state is transferred and discovered within the hypertext responses. Just like the browser needs to know about HTML, images, sound files, etc., a RESTful client will need to know how to parse the results of resolving a resource reference. However, the entire process is simple, constrained, scalable and flexible -- exactly the properties we want from a networked software system.

Many people build "RESTful" systems that require the clients to know beforehand what each level in a URL means. Should the information get reorganized on the server side, clients of those systems will break. Clients that truly embody HATEOS are more loosely-coupled from the servers they communicate with.

Looking Forward

We struggle daily to solve the problems of rapidly changing domains, technologies, customer demands and actionable knowledge. We spend too much time writing software to link what we learn to what we know. Objects and databases have not kept pace with the changes we experience. We need a new way of looking at the information we produce and consume that is extensible and less fragile than the solutions of the past. We need technology to help us form consensus. We should not have to achieve consensus in the form of common models before we can use our technologies.

In this article, we have introduced the series and have begun to look at how REST and Web technologies can serve as the basis of a new information-oriented architecture. We have established a naming scheme that allows us to unify references to all manner of content, services and documents. Clients can leverage the freedom to negotiate information into the form they want. As they resolve references, they can discover new content connected through new relationships.

This architectural style and the technologies surrounding the Semantic Web combine nicely to create powerful, scalable, flexible software systems. Their capacity to create Webs of Data will have as much impact on our lives as the Web has already had. This will be an information systems revolution that will turn much of what we know on its head. It will not only reduce the cost of data integration, but it will enable new business capabilities we can only begin to imagine.

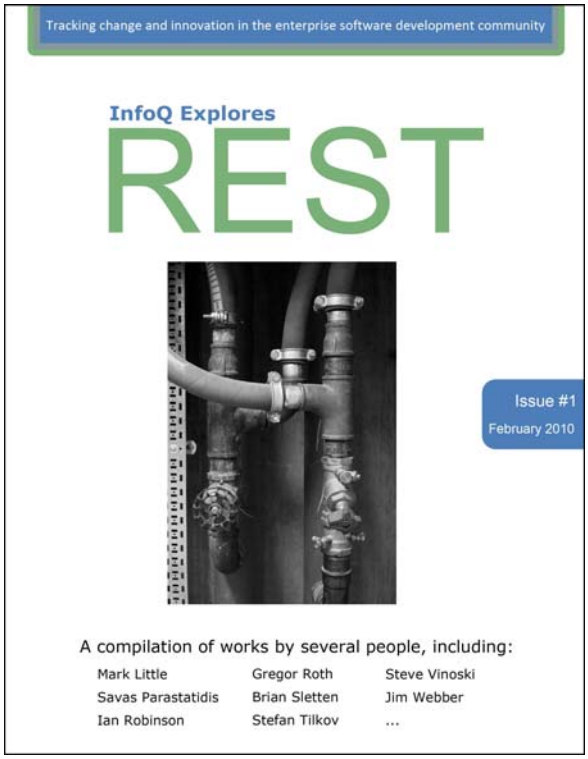
We are moving into a world where information can be connected and used regardless of whether it is contained in documents, databases or is returned as the results of a RESTful service. We will be able to discover content and connect it to what we already know. We will be able to surface the data currently hidden behind databases, spreadsheets, reports and other silos. Not only will we gain access to this information, we will be able to consume it in the ways we want to.

This is one of the main, modest goals of the Semantic Web. Achieving it, as we are now able to do, is starting to change everything.

Link: http://www.infoq.com/articles/roa-rest-of-rest

Related Contents：

- JavaOne Semantic Web Panel
- Cool URIs in a RESTful World
- A Comparative Clarification: Microformats vs. RDF
- The Semantic Web and Ontological Technologies Continue to Expand
- SPARQL Update to Complete RESTful SOA Scenario



InfoQ Explores: **REST**

Issue #1, March 2010

Chief Editor: Ryan Slobojan

Editors: Floyd Marinescu, Kevin Huo, Liu Shen

Feedback: ryan@infoq.com

Submit Articles: editors@infoq.com

InfoQueue
Enterprise Software Development Community