What if the Web Were not RESTful?

Cornelia Davis
EMC Corporation
Santa Barbara, CA
+1 805 560 9039
cornelia.davis@emc.com

ABSTRACT

So-called RESTful services are in widespread use both on the Web, and increasingly, in large enterprises. We say "so-called" because in reality, most of these services are not very RESTful. Those active in the REST community know well where these interfaces fail to meet REST principles, however, true understanding remains only in this relatively small community. Unfortunately, the result is a set of interfaces that are ultimately limited in their use, and the deficiencies are not usually recognized until it is too late to make the necessary changes. Our experience has shown that individuals are not being deliberately neglectful, rather, they simply do not know what they do not know. Everyone thinks they "get REST"; after all, using HTTP to move XML or JSON payloads over the network is very simple. We have found that most individuals begin to understand the nuances of REST when they are explained and they almost always ask for resources that further explain these concepts. There most certainly are materials available, however the best ones are rather substantial in size lessening the chance that many people will read them.

In this paper we take a fresh approach to explaining the core principles of REST, by describing a World Wide Web that fails to meet these tenets. We look at each key element, resource orientation, the uniform interface, media types and hyperlinking, and imagine the consequences of not abiding by the REST architectural style on the end user or tools developer of the Web. We then do a similar analysis in the context of Web services and programmatic consumers, reexamining each REST characteristic, describing common mistakes and suggesting improvements. We have found that in discussions, the analogy of the World Wide Web has been very effective at explaining REST.

Categories and Subject Descriptors

C2.2 [**Network Protocols**] Applications. C.2.4 [**Distributed Systems**] Client/server, Distributed applications. D.2.11 [**Software Architectures**]: Patterns, Languages, Service-oriented architecture. D.3.3 [**Language Constructs and Features**]: Patterns, Frameworks. D.2.3 [**Coding Tools and Techniques**]: Standards.

General Terms

Performance, Design, Reliability, Standardization, Theory

Keywords

REST, Architecture, Hypermedia, Statelessness, Resource-orientation, Media-types.

1. Introduction

The last several years have seen a significant increase in the popularity of "RESTful Web Services", both on the World Wide Web and in enterprise environments. And while these services have undoubtedly made the functionality of many systems more accessible than they would otherwise have been, there remains a significant divide between what these services do offer and what they would bring if they truly were RESTful. As evangelists for RESTful services within a large enterprise that produces many different software products, we have found that the primary reason for the deficiencies is a lack of understanding of what the REST architecture is, though most architects and developers believe they have a solid grasp. While there certainly are several outstanding references on the subject [6, 19, 24], what is lacking is a more abbreviated piece that helps the individual grasp the main concepts and motivates them to make the investment in reading these more thorough texts.

When presenting the tenets of REST and the value they bring to a solution, we have found explaining them in the context of the familiar World Wide Web is very effective. People use the Web every day for a broad range of things, from finding a good dry cleaner, to ordering the latest CD of their favorite artist from Amazon, to placing their takeout lunch order at the local deli, and while they are very familiar with the interactions required to accomplish those things, they do not understand how the REST architectural style has made these things possible. What we aim to do in the first part of this paper is explain the main elements of REST by exposing them through these familiar experiences. In the second part we transition to web services, specifically focusing on common deviations from the REST tenets; for each of these we make suggestions on how to offer required capabilities in a RESTful style and what the advantages brought.

2. Tenets of REST

While, at the very highest level, various works on the subject may present slightly different lists [13, 22], upon deeper inspection most agree on the following four fundamental principles:

1. Resource orientation and addressability: The central entity of a RESTful system is a resource.
2. Use of a [sufficiently rich] uniform interface: The set of ways that you can interact with a resource.
3. Resource representations and media types: The means for presenting resources.
4. Hypermedia as the Engine of Application State: Application flow is primarily (only?) through navigation of hyperlinks found in resource representations.

Another principle sometimes called out separately, stateless interactions, surfaces in several of the items called out above, and will be addressed within those contexts.

We now examine and explain each of these principles from the perspective of their impact in the World Wide Web today.

3. The RESTful Web

3.1 Resource Orientation and Addressability

This first tenet says that all interesting bits of information should be logically encapsulated in a *resource* that must be accessible via a stable identifier. On today's Web, resources range from documents (i.e. an article on the latest world cup rivalry or an photo snapped at the latest match), to the description of a car for sale, and an HTTP URL may be used to retrieve a resource representation. Almost twenty years of the World Wide Web have made this concept so familiar that it is nearly impossible to imagine a Web that is not resource-oriented. We are all accustomed to Web browsers and the notion that when we surf we are accessing one resource after another via the resource URL. The back button on our browser is an integral part of our Web-surfing experience.

Unfortunately, (and fortunately for the purposes of our discussion), recent programming practices have taken a Web that has long followed this principle and threatens to break it [1, 21]. The general design of these applications harkens back to the client/server architectures of the 1980s and 1990s with a JavaScript client running in the browser making opaque calls to a server endpoint to retrieve information, which is then processed in the JavaScript client for display. It is true that the call to the server is over HTTP, typically using the XMLHTTPRequest object [23], but the style is usually RPC tunneled over HTTP. When a Web user accesses such a program (the popular agile management software, Version One, for example), the URL in the browser remains fixed as they navigate through the application.

The consequences to such programming practices are damaging. While the display in the browser may appear to present a resource, the resource is actually hidden from various agents that depend on the resource orientation and addressability principle of REST. First, the user cannot bookmark the resource. All bookmarks will read the same, regardless of what is being displayed within the application. Second, Web caching is also broken. Between the origin server and the browser sit numerous layers, any of which can cache content, and the index for the cache is the URL for the resource. Even the browser acts as a cache, yet in this case, the browser sees only the single URL. When the caching infrastructure of the Web is broken, the end result is a significant degradation of the speed of rendering pages. And finally, another foundational piece of the web is also made impossible without resource orientation and addressability and that is the search index. An index requires a unit of index with a key for this unit; without addressable resources a search index would be impossible.

Without resource-orientation and addressability, today's World Wide Web could not exist.

3.2 Uniform Interface

Once we have a resource model and addressing scheme, the REST architectural style calls for the definition of a fixed set of operations with agreed upon semantics, which may be levied against those resources. The operations of the uniform interface[1] are defined in such a way that they serve specific needs for the infrastructure as a whole. To make this concept concrete, again, we look at the World Wide Web, where HTTP [5] is the uniform interface. HTTP defines the verbs GET, PUT, DELETE, POST, PATCH [4], OPTIONS and HEAD and places certain requirements on these actions; Table 1 summarizes these.

GET	Retrieve a resource representation	Safe, Idempotent, Cacheable
HEAD	Retrieve resource headers	Safe, Idempotent
OPTIONS	Used to get the list of verbs supported by this resource	Safe, Idempotent
PUT	Replace the resource	Idempotent
DELETE	Delete the resource	Idempotent
PATCH	Update the state of a resource	--
POST	Another operation on a resource Often used for resource creation	--[2]

Table 1 - Operations in the HTTP Uniform Interface

It is our experience that these characteristics are very often misunderstood; we explain them here.

3.2.1 Safe

An operation is safe if it may be executed with no-side effect to the resource state; GET, OPTIONS and HEAD are safe operations. Let's look at two scenarios where operation safety provides a great service to users on the World Wide Web.

First, consider how search is enabled on the World Wide Web. The search index for the web is created via a programmatic agent that accesses web resources to extract the tokens that become the entries in the index. Resource representations are accessed by issuing a GET to the resource URL. If GET were an unsafe operation, the act of indexing a resource could change the resource itself, a very negative consequence indeed.

[1] While the term "uniform interface" was used in Roy Fielding's seminal work [6] to refer to a broader set of constraints, its use

[2] POST responses may, in fact, be cached when used with certain control headers, however, this is possible in only limited cases and is rarely used.

A second scenario that depends upon the safety of GET is one where the browser invokes a URL that is embedded within the current resource representation, on behalf of the user, without the user explicitly requesting that access. One of the most common cases occurs when an HTML page contains an embedded image. When the browser receives such a resource representation it finds an `<img>` tag containing a `src` attribute with a URL to the resource that is the embedded image. To render the original Web page, the browser invokes a separate GET request to obtain the secondary resource for inclusion.

In another example, the HTML 5 draft specification [11] defines a general linking capability that allows related resources to be annotated with special semantics. One of those link types signals the browser that the referenced resource may be *prefetched*; that is, the resource representation may be retrieved before the user requests it. This type of link is used to identify resources that are likely to be requested next, and the prefetch may result in an increase in user perceived performance.

Web indexing and the described browser behaviors both depend on the safety of the GET operation.

3.2.2 Idempotent

An operation is idempotent if it may be executed one or more times with the same result. GET, OPTIONS, HEAD, PUT and DELETE are all idempotent. To help understand this concept, we again consider an example on the document Web that every user has experienced.

The first is an experience that every Web surfer has had. We are browsing the Web, click on a link and the Web browser initiates access to the resource, indicating "work in progress" through some type of status indicator. After 10 or 15 seconds (or, if you are particularly impatient, perhaps only 5) you assume some problem is interfering with the access to the resource, you click the "stop" button, click to access the resource again and this time it is displayed in only a few seconds. You are able to execute this failure recovery algorithm because you do not expect there to be any side effects of accessing a resource. By contrast, there are times when you would not execute this algorithm, for example when completing a purchase on some ecommerce site. If the browser hangs after clicking the "confirm order" button, your failure recovery algorithm would be more complex, involving checking your account to see whether the order had been completed and if not, only then would you feel confident clicking the "confirm order" button again. This example illustrates one of the significant benefits of an idempotent operation – the ability to simply retry an action if the first attempt appears to have failed. Not only can a human client initiate such actions, but so can a programmatic client.

A common misconception of idempotence is that it dictates no state change; this is not the case. The requirement is that any state change must result in the same resource state whether the operation is executed one or more than one time.

3.2.3 Cacheable

An operation is cacheable if the representation that is returned may be cached and used to serve the resource representation for future requests. GET and HEAD, are always cacheable, and when used with cache control and location headers, POST may also be cacheable. While the use of a cache is appropriately hidden from the end user on the Web, the benefit is significant. Instead of each request on the World Wide Web passing through numerous hops, all the way to the origin server for the resource, intermediaries can fulfill the access request with far less burden on the Internet. This is one of the main reasons for the horizontal scalability of the Web today and without this the World Wide Web would have been unable to serve the volume that it successfully serves today.

3.2.4 No Special Characteristics

Finally, it is important to understand the common operations of the HTTP interface that do not share any of these three characteristics and those are POST and PATCH. The PATCH method, which was standardized in March 2010, allows a resource state to be changed by presenting a set of changes to the resource. In most cases the resulting resource state depends on the starting state of the resource and hence the operation is not idempotent.

The POST operation is most often used on the document Web as a part of a form post. When such a form post results in the creation of a new resource, the new resource representation is returned and the cache-control and location headers are set, it may be cacheable. POST is not required to be used to only create new resources or return resource representations and in the absence of the aforementioned headers it is neither safe, idempotent nor cacheable.

HTTP is the uniform interface for the World Wide Web. It defines operations with certain characteristics that allow services such as search indexing and caching to be implemented. Without this uniform interface we would not be able to search the Web, nor would the Web have been able to grow unchecked as it has.

Without the uniform interface, today's World Wide Web could not exist.

3.3 Representations and Media Types

The third tenet is so significant that it exists in the name of the architectural style: REpresentational State Transfer (REST). This principal says that when a client is interacting with a resource, that it is not connected directly to the resource, rather, a resource representation acts as an abstraction between the client and the server. In the resource-oriented system we are examining, this principle is at the core of the "statelessness" constraint, which mandates that each service request be treated independently from any other request. That is, there are NO user sessions! The first fallacy of distributed computing [20] is that the network is reliable; stateless operations are the only way to build reliable systems in the face of these challenges.

Studies have shown [17] that the incidence of routing failure on the Internet is in the range of 1.5 to 3.4%, and while this research was conducted more than a decade ago, anecdotal evidence suggests these numbers remain accurate. Furthermore, with tens of thousands of machines serving a large ecommerce site, there is also a significant chance that one of those servers will fail at any given moment. If the software that allowed you to browse the Web site and make a purchase necessitated the stability of a single network connection to a single server over the course of your entire shopping experience, a significant number of purchases would never be made.

By allowing each request to stand on its own, which requires it carry all context for the operation within the request itself, the message can traverse the internet over whatever path is available and most optimal at that moment, and that any of the servers in the farm can respond to the request. The result is reliability over an unreliable network of unreliable servers.

When we consider the resource representations being transferred, standardization of the format is another critical factor. HTML was the first wildly successful resource representation format on the Web and its existence was directly responsible for the emergence of the Web browser. All Web sites that served HTML pages were immediately accessible using a Web browser, and that standardized format became a minimal barrier for entry (along with HTTP support) into the World Wide Web.

But we must also look deeper into how this user agent, the browser, knows how to behave for different resource representations (HTML vs. an image, for example). For example, how does the browser know which operation to invoke on behalf of the user. That is, how does the browser know to issue a GET request when I click on one hyperlink versus a POST request when I click on a button? The answer is that each form of hyperlink presented in a resource representation is typed, and the semantics of the link types are standardized; the media type scopes the set of link relations. The browser, then, implements behavior consistent with those agreed upon semantics. For example, the HTML media type defines link relations including the anchor (`<a>`) and the HTML form (`<form>`) to correspond to GET and POST operations, respectively.

Without the abstraction of the resource representation, the underlying reliability of the Internet would render the Web unusable. If it were the case that automobiles had a 3% chance of failure for any given outing, their ubiquity would be lacking; the use of the Web would be no different. Without format standards, each Web site would potentially require use of a customized client, effectively halting the ability for a client to surf the Web without barriers.

Without representational state transfer and standardized media types, today's World Wide Web could not exist.

3.4 The Hypermedia Constraint

The final REST tenet that we cover here states that resource representations must carry hyperlinks. A purchasing experience on an ecommerce site again provides a familiar context in which to explore the subtleties. Let's say you have just heard your first Lyle Lovett song and are eager to buy one of his CDs. You access your favorite ecommerce site and type "Lyle Lovett" into the search box, you submit, and a page displaying his CD catalog is returned. From this listing you drill down to see the details of any one of his CDs, you further drill down to read customer reviews, and eventually you place a CD into your shopping cart. You finalize your purchase by submitting an order. This entire process is driven by you invoking one HTTP operation after another, all by simply clicking on a "widget" in the current resource display.

If resources did not carry hyperlinks, both to other, related resources and to actions driving the application state, the shopping experience would be quite different. You might begin the experience by typing in the URL of the shopping site, say `http://myshoppingsite.com`. The Web page would present instructions on how to craft a URL to execute a search; it would be up to you to create the URL that expressed the collection of CDs associated with "Lyle Lovett"; this URL might look something like the following:

http://myshoppingsite.com/collections?artist=Lyle%20Lovett

Note that you would be responsible for following the rules for URL creation and URL encoding.

When the list of CDs was displayed and you wished to look at the details for, say, the "Pontiac" CD, you would again be responsible for crafting the URL to the resource. This might involve typing in an ID number for the CD that could be long and complex such as:

http://myshoppingsite.com/CDs/24215ae5606d2c4119e4

When you want to add this CD to your shopping cart, again, you must craft a URL, a request body, and then submit.

This step also exposes a more subtle aspect of the hypermedia constraint. On the read-only Web the client is only issuing GET requests[3] and therefore each hyperlink will be invoked with a GET operation. In the purchasing scenario we've just described, however, the client is also *contributing* information using an operation such as POST. This means that in the awkward, non-hyperlinked purchasing scenario we describe above, the user is not only responsible for crafting hyperlinks, they are also responsible for choosing the HTTP operation and formatting the request body. Recall from the previous section that the media type scopes link relations. Here we emphasize that it is the link relation (i.e. the anchor versus the form) that defines the appropriate HTTP verb, NOT the target URL.

While the picture we painted here, with shoppers crafting URLs and choosing HTTP operations may seem absurd, if the Web only had the three other REST tenets but did not have the hypermedia constraint, this would be the experience. Furthermore, as we shall see in the second half of this paper, most so-called RESTful services completely neglect this principle, requiring the absurd of the service client.

Without the hypermedia constraint, today's World Wide Web could not exist.

4. RESTful Web Services

In the first half of this paper we tried to imagine a World Wide Web that was not RESTful, examining each of the key tenets in turn, and reaching the conclusion that every one was essential. By extension, we assert that all four tenets are equally critical for RESTful Web Services, yet in most cases, one or more principle is lacking. In this section we revisit each core REST principle, examine some of the common errors made in the design of so-called RESTful Web Services and suggest remediation for each.

4.1 Resource Orientation and Addressability

Most RESTful Web Services available today project resources with representations accessible via URL. Specifications such as Jax-RS [10], and development frameworks such as Apache CXF [27] and the Spring Framework [12] present resources as first-class entities and provide allow URLs for those resources to be assigned using things such as URI templates [7]. There is, however, an area that could benefit from an approach that more fully exploits the benefits of the REST architectural style, and that is with search interfaces.

Many search interfaces included in RESTful services are defined so that a search query is POSTed to a particular URL and the result of that query is the resource representation returned. With such a design, the set of Lyle Lovett CDs will share the same

[3] HEAD and OPTIONS are also read requests, though in the context of this example they are not a factor.

URL as the set of Cesaria Evora CDs; that is, these collections are NOT resources exposed through the service. While not being able to bookmark this collection (because the resource does not exist) may be less of a concern with a programmatic client, the lack of additional capabilities afforded to addressable resources poses a significant disadvantage. The collection cannot be monitored for changes via eTags and it cannot be cached. Semantics such as those offered by RFC 5005, Feed Paging and Archiving [15], which defines a means for paging through large collections, cannot be leveraged because RFC 5005 requires the definition of subsets of the collection as resources.

A search interface of this design first, reflects a predisposition for an RPC-centric perspective, and second, exposes an implementation detail through the interface. The fact that a service finds the collection of Lyle Lovett CDs via a query is a detail that is not of interest to the consumer. Instead, the Web Service should project the resources of interest, that is, CD collections for individual artists, via addressable resources.

As we shall emphasize in the section on the hypermedia constraint below, the client should not be responsible for crafting such a URL, thus we must consider patterns for providing these; we suggest two here.

First let us return to our analogy of the shopping experience on the World Wide Web. When I initiate my search for Lyle Lovett CDs, I enter the string "Lyle Lovett" into a Web form that is then POSTed to the ecommerce site. Is this not exactly the pattern that we are discouraging? Not quite. When that search string is posted to the site, rather than returning the results of the search in the POST response, the site returns a redirect with a URL that addresses the desired collection resource; the URL in my browser now will read something like http://myshoppingsite.com/collections?artist=Lyle%20Lovett. While this redirect pattern is not uncommon on the Web today, it is almost entirely neglected in Web Services. What is required of RESTful Web Services to implement this pattern is a resource naming service that accepts inputs via a POST and returns the URL for the resultant resource, with a 3xx status code. The service must then also service GET requests against those collection resource URLs.

A second pattern involved in a RESTful search is the use of OpenSearch [32]. OpenSearch defines a mechanism whereby the translation from search terms to a URL is precisely defined so that programmatic clients can perform the naming operations. OpenSearch provides a level of abstraction where URL templates are declared, allowing an interpreter, rather than hard coded logic, to be used for resource naming functions. In fact, the resource naming service described in the first pattern could be implemented as an OpenSearch client application.

There are popular Web services available today that properly follow the resource-orientation and addressability tenet, such as the Google Data APIs [29], however, found few RESTful services that address the resource naming concern[4].

4.2 Uniform Interface

As with resource orientation and addressability, the uniform interface constraint is most often fairly well followed in RESTful services. There has been sufficient coverage [3, 8, 14] of egregious errors previously made, that the incidence of flagrant violations has reduced significantly. We have observed, however, two common errors persist.

The first is the misuse of PUT. As you will recall from earlier sections in this paper, PUT is required to be idempotent; that is, PUT may be executed once, or more than once with the same result. When PUT is used to change *part* of the state of a resource there is a significant risk that the operation will no longer be idempotent, and in some cases the result may even be an inconsistent resource state. Let us examine this with a specific example.

Consider a service that exposes invoice resources. The resource includes purchaser information, the items purchased, the sales tax levied and the invoice total. The following is an example resource representation served with a GET to http://example.com/invoices/1.

```
<invoice xmlns="http://ns.example.com/ordering">
    <purchaser>
        <name>The Client</name>
        …
    </purchaser>
    <items>
        <item>
            <partno>123</partno>
            <quantity>1</quantity>
            <priceper>10</priceper>
            <total>10</total>
        </item>
    </items>
    <tax></tax>
    <invoicetotal>10.87</invoicetotal>
</invoice>
```

Now let us consider a client that calculates the sales tax on an invoice by obtaining the resource representation via GET, performing the calculation, and issuing a PUT back to the resource with a partial representation that updates the resource state; for example, a PUT to http://example.com/invoices/1 with only the following body will update only the tax portion of the resource state.

```
<tax>0.87</tax>
```

Assuming no other clients are accessing the resource, issuing this PUT repeatedly seems to be idempotent, however, let us now introduce a second client that makes changes to the line items in the invoice. This client could issue a PUT to http://example.com/invoices/1 with the body shown below, replacing only the items on the invoice.

```
<items>
    <item>
        <partno>123</partno>
        <quantity>1000</quantity>
        <priceper>10</priceper>
        <total>10000</total>
    </item>
</items>
```

If the tax-calculating client issues a PUT based on the original resource state and never receives a response, it or an intermediary may retry the PUT, assuming idempotence. If the second client's PUT is issued between the two PUT attempts from the first, the affect of executing those two PUT operations will differ. Worse, the resource state will be inconsistent with a subtotal of $10,000 and a tax calculation of only 87 cents.

There are, of course, various ways in which this problem may be avoided [2]; we most often recommend using PATCH. Using

[4] This is actually more a concern for the hypermedia constraint.

PATCH without conditional request headers cannot be assumed idempotent, and therefore an automated client or intermediary cannot simply retry failed requests. Using PATCH with conditional request headers will further help address concurrency control problems such as that illustrated with the above example.

A second point of frequent debate with respect to the uniform interface is the frequency with which POST is used relative to the other HTTP methods. Often, when we see an interface with a great many POST operations we find that the so-called RESTful services are simply tunneling RPC requests over HTTP; existing implementation object oriented models have been exposed almost directly with each method call exposed via HTTP POST. Aside from these cases, which expose a significant deficiency in the understanding of REST, there are several legitimate reasons for preferring POST to other side-effecting operations such as PUT, PATCH or DELETE.

Some clients, such as Adobe Flex, only support dispatching GET and POST HTTP requests, and sometimes firewalls are configured to only allow GET and POST requests to pass. These scenarios have tempted some of our product development groups to avoid creating a services design with DELETE or PUT operations, however, we routinely advise them differently. Rather than constraining the services design for restricted consumption scenarios and with that giving up some of the benefits that can be realized with idempotent operations, we encourage the development of services that use the full range of HTTP verbs available. Clients that can issue HTTP requests beyond GET and POST can then take advantage of the benefits these other requests afford.

How, then, can these DELETE, PUT and PATCH operations be leveraged in deployment scenarios where only GET and POST are permitted? In this case we recommend the use of the x-HTTP-method-override header where the client issues a POST request and carries a value in this header to indicate the required operation. The RESTful services, then, must have a filter deployed before them that intercepts POST requests and forwards on to the services implementation a request with the HTTP method given in that header. Many REST service frameworks such as Jersey [31], CXF [27] and WCF [33] provide such filter implementations. Of course, when POST and the method override header are used, the benefits of using PUT and DELETE are sacrificed, however, those options remain available in the services and other clients may benefit.

4.3 Representations and Media Types

While we have found that most of the RESTful services we have examined are stateless, there are two commonly made mistakes that come with significant negative consequences.

Particularly when coming from development teams that have a legacy in client/server architectures, we have all too often found RESTful Web services made available along with a client-side library. The client-side library hides the RESTful service from the client-side programmer entirely, taking on the task of creating and dispatching service requests, and unmarshalling any responses into client-side objects. While the existence of libraries on the client-side are not evidence enough that the services are not RESTful, after all, even a generic HTTP-REST client surely uses libraries for HTTP interactions, in our experience, treating the resource representation as nothing more than a serialization of objects almost always results in deficiencies in the service interfaces. For example, when designed this way, resources are often not presented at the right level of granularity. Let us consider a scenario where a service provides the ability to create and read medical records; this example roughly describes the requirements of the XDS specification [30] from the Integrating the Healthcare Enterprise Consortium. A medical record consists of folders, documents and relationships (between folders and documents). Various collection resources are exposed on read, projecting folder and document resources in different combinations and individually, calling for a relatively granular resource model for read. When medical records are written, however, there are requirements that the whole medical record be treated as an atomic resource, where either all enclosed folders, documents and relationships are written, or none are. If the services developer simply uses HTTP as a transport protocol, they very likely will expose folder, document and relationship resources individually, both for read and for write. The result is that the developer writing a client to produce medical records will start looking into distributed transaction patterns, essentially rendering the so called RESTful service stateful.

We note that with the increasing popularity of application/json as a media type, this practice of using so-called RESTful Web services primarily as an object transport protocol is becoming more widespread rather than less. When consulting with teams building RESTful services we always emphasize a resources-first design paradigm.

The second common error in this aspect of RESTful service design comes when media types are not used to scope the processing semantics of the resource types. This is a complex and nuanced topic that is perhaps best understood by examining a case where this was done correctly; the Atom Syndication Format, RFC 4287 [16] gives us this exemplar.

The Atom Syndication Format (henceforth referred to simply as "Atom") is defined with the media type application/atom+xml. The format defines both a schema and the context in which link relations can be understood. This allows for the construction of a client that "speaks Atom" and many such clients exist [28]. The Atom format was carefully defined to allow extensions and the Atom Publishing Protocol, RFC 5023 [9], is one that illustrates a very key point. The Atom Publishing Protocol (henceforth referred to simply as AtomPub) defines a link relation with the rel value "edit". The semantics of this link relation are that a client may issue a DELETE to the href present in the link, resulting in the deletion of the containing resource. The "edit" link may also be used to update the containing resource state through a GET, client-side modification of the resultant resource representation, followed by a PUT, again to the URL found in the href of the edit-link. A client may be written that implements behaviors specific to the semantics defined for AtomPub.

In a moment we will turn our attention back to the hypermedia constraint, yet, to facilitate another example for the role of media types in RESTful services, for the moment we assume resource representations carry hyperlinks. Often we find services that present resource representations using XML and a common practice is to include an href attribute on any arbitrary XML element. Consider, for example, the following representation of a storage pool resource:

```
<StoragePoolCapacityData>
    <Name>Pool for ABC00745 storage system</Name>
    <Id>C24340683</Id>
    <StorageArray
url="http://example.com/datacenter/storagearrays/1
23"/>
    <Type>RAID Group</Type>
    <UsableSpace>1728966445056</UsableSpace>
    ...
</StoragePoolCapacityData>
```

While this resource representation does include URLs, noting that they may be indicated through a variety of means (i.e. using an attribute named url instead of the more usual href) they may be difficult for the client to find. What we have most often found is that this knowledge is encoded directly into the client. Further, the details of which operations can be issued against a URL are also implicitly encoded in the client side logic. If, instead of using a generic XML media type such as text/xml, a specific media type were created for this format, when the client obtained a representation of this media type it would know just how to find and use the embedded hyperlinks. This is exactly what was done with the Atom media type, application/atom+xml.

When advising product groups on RESTful design we often recommend the use of Atom as the media type. The basic model of individual entities and collections can be almost ubiquitously applied. The set of extensions to Atom such as Atom Pub [9] and Feed Paging and Archiving [15] are also very useful. But perhaps the most compelling reason for following this guidance is that standard Atom clients [28] may be used to access these representations, which can be an effective developer tool.

Following this guidance the above example would be as follows.

```
<atom:entry
xmlns:atom="http://www.w3.org/2005/Atom">
    <atom:title>
      Pool for ABC00745 storage system
    </atom:title>
    <atom:id>C24340683</atom:id>
    <atom:updated>
      2012-02-17T06:59:04Z
    </atom:updated>
    <atom:summary>
      Pool data for ABC00745 storage system
    </atom:summary>
    <atom:link
  rel="http://ns.example.com/rels/containingArray"
  href="http://example.com/it/storagearrays/123"/>
    <Type>RAID Group</Type>
    <UsableSpace>1728966445056</UsableSpace>
    ...
</atom:entry>
```

Because Atom explicitly allows for extensions, however, the media type no longer defines the exact set of link relations that may be found. This means that a client built specifically to handle storage system resources is forced to encode semantics that are not explicitly tied to the media type. We are currently exploring the use of media type profiles to address this challenge [18, 26]. While further work is needed here, in no way does it diminish the importance of the media type in the REST architectural style.

When RESTful Web services do not properly use media types by ignoring that they carry with them processing semantics, including hyperlink behaviors, the client implementation will implicitly include out of band knowledge, resulting in a tightly coupled design. Instead, if the service uses the media type as the to encapsulate not only format but behavioral semantics, then general purpose clients or reusable client components specific to that media type may be built.

4.4 The Hypermedia Constraint

The picture we painted earlier, of a Web without hyperlinks, was indeed absurd; in fact links are arguably the very strands of the World Wide Web. Yet, the vast majority of the so-called RESTful Web services we have examined are completely devoid of hyperlinks. Even the Google Data APIs, with their widespread use, for example, require the client craft URLs when navigating through resource representations. The very activities that we would never impose on a human consumer of the World Wide Web are thrust upon the client developer of the programmatic Web with regularity.

When we force a client into crafting the URLs that they will invoke, we are first, burdening them with a responsibility that is best achieved by the service itself. Secondarily, the URL formats become fixed; the service may not change the format of resource URLs without breaking existing clients. And finally, whenever new resource types are introduced, the client must first be upgraded before the new, implicit relationships can be leveraged.

Resource representations lacking in hyperlinks result in further tight coupling between the client and server because the latter is not projecting the valid application state transitions. Instead, the client must encode all of the application logic, at the expense of significant complexity. Consider an example where an application resource includes an Atom link relation with rel value “edit” only when the user requesting the resource has the privileges to make changes to it. When that link relation is not present, the client needn’t attempt such an access; when that link relation is present it indicates that a modification of the resource is a valid next step for the application. If services return resource representations completely devoid of hyperlinks, the client has no alternative than to codify the application state transition logic, with complex recovery logic, directly into their implementation.

One other recommendation we often make is that atom link relations be used in any type of XML resource representations. As described in the previous section, media types scope the format and semantics of link relations and while in theory this means that links can come in any form, much as HTML carries links of many different forms, pragmatically, it is easier on a client developer if they have a uniform way of finding and interacting with hyperlinks.

When we force a client into crafting the URLs or embedding complex state transition logic, we are introducing a coupling between the client and service that is completely unnecessary. Instead, the service should include hyperlinks, both to related resources, which are analogous to HTML links such as anchors and images, and to POST URLs, which are analogous to HTML forms. This allows the service to be upgraded without affecting existing clients, and if done right, may even allow an application to be upgraded simply through the upgrade of that service.

Of the four tenets of the REST architectural style, the hypermedia constraint has proven to be the most difficult to fulfill, yet the resultant benefits are as significant for Web services and programmatic clients as they are for the World Wide Web.

5. Conclusions

Research and writing on the REST architectural style and RESTful services has been ongoing for more than a decade and a strong community of REST practitioners has emerged. Unfortunately, this community is extremely small relative to the rapidly expanding set of individuals and organizations engaged in building Web services that should be RESTful. The recent popularizing of cloud-based computing and services has served to only increase the relevance of REST and the importance of the tenets of the REST architectural style. Despite this, widespread understanding of these principles remains lacking. The concepts are more complex than most individuals believe, and the benefits of following certain practices and the disadvantages posed when they are not, are most often overlooked. By presenting each of the key principles of the REST architectural style in the context of interactions that are commonplace in the World Wide Web, we hope to expose the reader to these subtleties and encourage them to think more deeply and investigate more thoroughly.

6. References

[1] Carlos, Sean, Eight Reasons to Avoid Flash (or Silverlight) Like the Plague When Designing a Website. Blog post, June 2009. See http://antezeta.com/news/flash-problems.

[2] Davis, Cornelia, More on PUT Idempotency. Blog post, February 2011. See http://corneliadavis.com/blog/2011/02/19 /more-on-put-idempotency/.

[3] De hOra, Bill, Hey, I'm back. Blog post, October 2005. See http://www.dehora.net/journal/2005/10/hey_im_back.html.

[4] Dusseault, Lisa, James M. Snell, PATCH Method for HTTP, Internet RFC 5789, March 2010.

[5] Fielding, Roy Thomas, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, Tim Berners-Lee, Hypertext Transfer Protocol — HTTP/1.1, Internet RFC 2616, June 1999.

[6] Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

[7] Gregorio, J., R. Fielding, M. Hadley, M.Nottingham, D. Orchard, URI Template, Internet Draft draft-gregorio-uritemplate-08, January 2012.

[8] Gregorio, J., I'm sorry, I can't kiss it and make it better. Blog post, May 2005. See http://bitworking.org/news /I_m_sorry__I_can_t_kiss_it_and_make_it_better_.

[9] Gregorio, J. and B. de hOra. The Atom Publishing Protocol. October 2007. See http://tools.ietf.org/html/rfc5023.

[10] Hadley, M. and P. Sandoz. JAX-RS: Java API for RESTful Web Services. September 2009. See http://jcp.org/en/jsr/detail?id=311.

[11] Hickson, Ian HTML5: A Vocabulary and Associated APIs for HTML and XHTML, World Wide Web Consortium, Working Draft WD-html5-20120329, March 2012

[12] Johnson, R., et. al. Spring Framework Reference Documentation 3.0. 2004-2010. See http://static.springsource.org/ spring/docs/3.0.x/spring-framework-reference/html/.

[13] Mahemoff, Michael, Ajax Design Patterns, O'Reilly & Assoiciates, Sebastopol, California, July 2006, 978-0596101880.

[14] Moertel, Tom, Google Web Accelerator vs. unsafe linking: Round Two! Blog post, October 2005. See http://blog.moertel.com/articles/2005/10/25/google-web-accelerator-vs-unsafe-linking-round-two.

[15] Nottingham, Mark, Feed Paging and Archiving, Internet RFC 5005, September 2007.

[16] Nottingham, M. and R. Sayre. The Atom Syndication Format. December 2005. See http://tools.ietf.org/html/rfc4287.

[17] Paxson, V, End-to-end routing behavior in the internet. IEEE/ACM Transactions on Networking, 5(5):601–615, October 1997.

[18] Raggett, Dave, Arnaud Le Hors, Ian Jacobs, HTML 4.01 Specification, World Wide Web Consortium, Recommendation REC-html401-19991224, December 1999.

[19] Richardson, Leonard, Sam Ruby, RESTful Web Services, O'Reilly & Associates, Sebastopol, California, May 2007, 0-596-52926-0.

[20] Rotem-Gal-Oz, Arnon, Fallacies of Distributed Computing Explained. Blog posted whitepaper. See http://www.rgoarchitects.com/Files/fallacies.pdf

[21] Tennison, Jeni, Hash URIs. Blog post, March 2011. See http://www.jenitennison.com/blog/node/154.

[22] Tilkov, Stefan, A Brief Introduction to REST, InfoQ, http://www.infoq.com/articles/rest-introduction, December 2007.

[23] van Kesteren, Anne, XMLHttpRequest Level 2, World Wide Web Consortium, Working Draft WD-XMLHttpRequest2-20100907, September 2010.

[24] Webber, Jim, Savas Parastatidis, Ian Robinson, REST in Practice: Hypermedia and Systems Architecture, O'Reilly & Associates, Sebastopol, California, September 2010, 978-0-596-80582-1.

[25] Wilde, Erik, Links in HTML. Blog post, October 2009. See http://dret.typepad.com/dretblog/2009/10/links-in-html.html.

[26] Wilde, Erik, Resource Profiles. Blog post, March 2012. http://dret.typepad.com/dretblog/2012/03/resource-profiles.html.

[27] Apache CXF: An Open Source Services Framework. See http://cxf.apache.org/.

[28] The AtomEnabled Directory: Client Software. See http://www.atomenabled.org/everyone/atomenabled/index.php?c=5.

[29] Google Data APIs. See http://code.google.com/apis /gdata/docs/directory.html.

[30] Integrating the Healthcare Enterprise Technical Frameworks. See http://www.ihe.net/Technical_Framework/.

[31] Jersey. Jax-RS Reference Implementation for building RESTful Services. http://jersey.java.net/.

[32] OpenSearch Specification, version 1.1. See http://www.opensearch.org/Specifications/OpenSearch/1.1.

[33] Windows Communication Foundation. See http://msdn.microsoft.com/en-us/netframework/aa663324.