RESTful Service

Todd Fredrich
Pearson eCollege

Table of Contents

Document History

Date	Version	Description
Feb 10, 2012	Draft	Initial draft version.
Apr 24, 2012	v1.0	Initial public (non-draft) version.
May 29, 2012	v1.1	Minor updates to correct misspellings and clarify wording after feedback from API Best Practices Task force.
Aug 2, 2013	v1.2	Updated versioning section. Additional minor corrections of misspellings, wording, etc.

Who Should Read This Document

This best-practices document is intended for developers who are interested in creating RESTful Web services that provide high reliability and consistency across multiple service suites. By following these guidelines, services are well positioned for rapid widespread public adoption by both internal and external clients.

The guidelines in this document are also appropriate for engineers who support services developed using these best practices. While their concerns may be focused on caching practices, proxy rules, monitoring, security, and such, this document may be useful as an overarching service documentation guide of sorts.

Additionally, management personnel may benefit from these guidelines by endeavoring to understand the effort required to create services that are publicly consumable that offer high levels of consistency across their service suites.

Introduction

There are numerous resources on best practices for creating RESTful web services (see the Resources section at the end of this document). Many of the available resources are conflicting, depending on when they were written. Plus, reading and comprehending several books on the subject in order to implement services "tomorrow" is not doable. In order to facilitate quick uptake and understanding of RESTful concepts, without requiring the reading of at least three to five books on the subject, this guide is meant to speed up the process—condensing REST best practices and conventions into just the high points with not a lot of discussion.

REST is more a collection of principles than it is a set of standards. Other than its over-arching six constraints, nothing is dictated. There are "best practices" and de-facto standards but those are constantly evolving—with religious battles waging continuously.

Designed to be brief, this document provides recommendations and some cookbook-style discussion on many of the common questions around REST and provides some short background information to offer support for effective creation of real-world, production-ready, consistent RESTful services. This document aggregates information available in other sources, adapting it with experience gained through hard knocks.

There is still considerable debate as to whether REST is better than SOAP (and vice versa), and perhaps there are still reasons to create SOAP services. While touching on SOAP, this document won't spend a lot of time discussing relative merits. Instead, because technology and the industry marches on, we will proceed with the assumption that leveraging REST is the current best practice for Web service creation.

The first section offers an overview of what REST is, its constraints, and what makes it unique. The second section supplies some quick tips as little reminders of REST service concepts. Later sections go more in depth to provide the Web service creator more support and discusses the nitty-gritty details of creating high-quality REST services capable of being publicly exposed in a production environment.

What is REST?

The REST architectural style describes six constraints which were originally communicated by Roy Fielding in his doctoral dissertation (see http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm). They define the basis of RESTful-style.

The six constraints are:

- Uniform Interface
- Stateless
- Cacheable
- Client-Server
- Layered System
- Code on Demand

A more detailed discussion of the constraints follows:

Uniform Interface

The uniform interface constraint defines the interface between clients and servers. It simplifies and decouples the application architecture, which enables each part to evolve independently. The four

guiding principles of the uniform interface are:

Resource-Based

Individual resources are identified in requests using URIs as resource identifiers. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server does not send its database, but rather, some HTML, XML, or JSON that represents some database records expressed, for instance, in Finnish and encoded in UTF-8, depending on the details of the request and the server implementation.

Manipulation of Resources Through Representations

When a client holds a representation of a resource, including any attached metadata, the client has enough information to modify or delete the resource on the server, provided it has permission to do so.

Self-descriptive Messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cache-ability.

Hypermedia as the Engine of Application State (HATEOAS)

Clients deliver state via body contents, query-string parameters, request headers, and the requested URI (the resource name). Services deliver state to clients via body content, response codes, and response headers. This is technically referred-to as hypermedia (or hyperlinks within hypertext).

Aside from the description above, HATEOS also means that, where necessary, links are contained in the returned body (or headers) to supply the URI for retrieval of the object itself or related objects. We'll talk about this in more detail later.

The uniform interface that any REST services must provide is fundamental to its design.

Stateless

As REST is an acronym for **RE**presentational **S**tate **T**ransfer, *statelessness* is key. Essentially, what this means is that the necessary state to handle the request is contained within the request itself, whether as part of the URI, query-string parameters, body, or headers. The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Then after the server does its processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status codes, and response body.

Most of us who have been in the industry for a while are accustomed to programming within a container which provides the concept of "session" which maintains state across multiple HTTP requests. In REST, the client must include all information for the server to fulfill the request, resending state as necessary if that state must span multiple requests. Statelessness enables greater scalability since the server does not have to maintain, update, or communicate that session state. Additionally, load balancers don't have to worry about session affinity in stateless systems.

So what's the difference between state and a resource? State, or *application state*, is that which the server cares about to fulfill a request—data necessary for the current session or request. A resource, or *resource state*, is the data that defines the resource representation—the data stored in the database, for instance. Consider *application state* to be data that could vary by client, and per request. *Resource*

state, on the other hand, is constant across every client who requests it.

Ever had back-button issues with a web application where it went AWOL at a certain point because it expected you to do things in a certain order? That's because it violated the *statelessness* principle. There are cases that don't honor the *statelessness* principle, such as three-legged OAuth, API call rate limiting, etc. However, make every effort to ensure that application state doesn't span multiple requests of your service(s).

Cacheable

As on the World Wide Web, clients can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Client–server

The uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface is not altered.

Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches. Layers may also enforce security policies.

Code on demand (optional)

Servers are able to temporarily extend or customize the functionality of a client by transferring logic to it that it can execute. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

Complying with these constraints, and thus conforming to the REST architectural style, will enable any kind of distributed hypermedia system to have desirable emergent properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

NOTE: The only optional constraint of REST architecture is *code on demand*. If a service violates any other constraint, it cannot strictly be referred to as RESTful.

REST Quick Tips

Whether it's technically RESTful or not (according to the six constraints mentioned above), here are a few recommended REST-like concepts that will result in better, more usable services:

Use HTTP Verbs to Mean Something

Any API consumer is capable of sending GET, POST, PUT, and DELETE verbs, and they greatly enhance the clarity of what a given request does. Also, GET requests *must* not change any underlying resource data. Measurements and tracking may still occur, which updates data, but not resource data identified by the URI.

Sensible Resource Names

Having sensible resource names or paths (e.g., /posts/23 instead of /api?type=posts&id=23) improves the clarity of what a given request does. Using URL query-string parameters is fantastic for filtering, but not for resource names.

Appropriate resource names provide context for a service request, increasing understandability of the service API. Resources are viewed hierarchically via their URI names, offering consumers a friendly, easily-understood hierarchy of resources to leverage in their applications.

Resource names should be nouns—avoid verbs as resource names. This makes things more clear. Use the HTTP methods to specify the verb portion of the request.

XML and JSON

Favor JSON support as the default, but unless the costs of offering both JSON and XML are staggering, offer both. Ideally, let consumers switch between them by just changing an extension from .xml to .json. In addition, for supporting AJAX-style user interfaces, a wrapped response is very helpful. Provide a wrapped response, either by default or for separate extensions, such as .wjson and .wxml to indicate the client is requesting a wrapped JSON or XML response (see Wrapped Responses below).

JSON in regards to a "standard" has very few requirements. Plus, those requirements are only syntactical in nature, not about content format or layout. In other words, the JSON response to a REST service call is very much part of the contract—not described in a standard. More about the JSON data format can be found at http://www.json.org/.

Regarding XML use in REST services, XML standards and conventions are really not in play other than to utilize syntactically correct tags and text. In particular, namespaces are not, nor should they be used in a RESTful service context. XML that is returned is more JSON like—simple and easy to read, without the schema and namespace details present—just data and links. If it ends up being more complex than this, see the first paragraph of this tip—the cost of XML will be staggering. In our experience few consumers use the XML responses anyway. This is the last 'gasp' before XML gets phased out entirely.

Create Fine-Grained Resources

When starting out, it's much easier to create APIs that mimic the underlying application domain or database architecture of your system. Eventually, you'll want aggregate services—services that utilize multiple underlying resources to reduce chattiness. But it's much easier to create larger resources later from individual resources than it is to create fine-grained or individual resources from larger aggregates. Make it easy on yourself and start with small easily defined resources, providing CRUD functionality on those. You can create those use-case-oriented, chattiness-reducing resources later.

Consider Connectedness

One of the principles of REST is connectedness—via hypermedia links. While services are still useful without them, APIs become more self-descriptive when links are returned in the response. At the very least, a 'self' reference informs clients how the data was or can be retrieved. Additionally, utilize the *Location* header to contain a link on resource creation via POST. For collections returned in a response that support pagination, 'first', 'last', 'next', and 'prev' links at a minimum are very helpful.

Definitions

Idempotence

Contrary to how it sounds, make no mistake, this term has no relation to certain areas of disfunction. From Wikipedia:

> *In computer science, the term idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times. This may have a different meaning depending on the context in which it is applied. In the case of methods or subroutine calls with side effects, for instance, it means that the modified state remains the same after the first call.*

From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result—operating much like a "setter" method in a programming language. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, read the caveat on DELETE in the *HTTP Verbs, DELETE* section below.

GET, HEAD, OPTIONS and TRACE methods are defined as idempotent also, since they are defined as safe. Read the section on safety below.

Safety

From Wikipedia:

> *Some methods (for example, HEAD, GET, OPTIONS and TRACE) are defined as safe, which means they are intended only for information retrieval and should not change the state of the server. In other words, they should not have side effects, beyond relatively harmless effects such as logging, caching, the serving of banner advertisements or incrementing a web counter. Making arbitrary GET requests without regard to the context of the application's state should therefore be considered safe.*

In short, safety means that calling the method does not cause side effects. Consequently, clients can make safe requests repeatedly without worrying about side effects on the server. This means that services must adhere to the safety definitions of GET, HEAD, OPTIONS and TRACE operations. Otherwise, besides being confusing to service consumers, it can cause problems for Web caching, search engines, and other automated agents—making unintended changes on the server.

By definition, safe operations are idempotent, since they produce the same result on the server.

Safe methods are implemented as read-only operations. However, safety does not mean that the server must return the same response every time.

HTTP Verbs

The HTTP verbs comprise a major portion of our "uniform interface" constraint and provide the action counterpart to the noun-based resource. The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, and DELETE. These correspond to create,

read, update, and delete (or CRUD) operations, respectively. There are a number of other verbs, too, but are utilized less frequently. Of those less-frequent verbs, OPTIONS and HEAD are used more often than others.

GET

The HTTP GET method is used to retrieve (or read) a representation of a resource. In the "happy" (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

Examples:
GET http://www.example.com/customers/12345
GET http://www.example.com/customers/12345/orders
GET http://www.example.com/buckets/sample

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered *safe*. That is, they can be called without risk of data modification or corruption—calling them once has the same effect as calling them10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.

PUT

PUT is most-often utilized for update capabilities, PUT-ing to a known resource URI with the request request body containing the newly-updated representation of the original resource.

However, PUT can also be used to create a resource when the resource ID is chosen by the client instead of by the server — in other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see *POST* below).

Examples:
PUT http://www.example.com/customers/12345
PUT http://www.example.com/customers/12345/orders/98765
PUT http://www.example.com/buckets/secret_stuff

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID. See the *Return Values* section below.

PUT is not a *safe* operation, in that it modifies (or creates) state on the server, but it is *idempotent*. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer *idempotent*. Sometimes that happens and it may be enough to document that the call is not *idempotent*. However, it's recommended to keep PUT requests *idempotent*. It is strongly recommended to use POST for non-idempotent requests.

POST

The POST verb is most-often utilized for creating new resources. In particular, it's used to create *subordinate* resources — that is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

Examples:
POST http://www.example.com/customers
POST http://www.example.com/customers/12345/orders

On successful creation, return HTTP status 201, returning a *Location* header with a link to the newly-created resource with the 201 HTTP status.

POST is neither *safe* or *idempotent*. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most likely result in two resources containing the same information.

PUT vs POST for Creation

In short, favor using POST for resource creation. Otherwise, use PUT when the client is in charge of deciding which URI (via its resource name or ID) the new resource will have. If the client knows what the resulting URI (or resource ID) will be, use PUT at that URI. Otherwise, use POST when the server or service is in charge of deciding the URI for the newly-created resource. In other words, when the client doesn't (or shouldn't) know what the resulting URI will be before creation, use POST to create the new resource.

DELETE

DELETE is pretty easy to understand. It is used to delete a resource identified by a URI.

Examples:
DELETE http://www.example.com/customers/12345
DELETE http://www.example.com/customers/12345/orders
DELETE http://www.example.com/buckets/sample

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (if it doesn't demand too much bandwidth), or a wrapped response (see *Return Values* below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

HTTP-spec-wise, DELETE operations are *idempotent*. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer *idempotent*. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This makes DELETE operations no longer idempotent, but is an appropriate compromise if resources are removed from the database instead of being simply marked as deleted.

Below is a table summarizing recommended return values of the primary HTTP methods in combination with the resource URIs:

HTTP Verb	/customers	/customers/{id}
GET	200 (OK), list of customers. Use pagination, sorting, and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	404 (Not Found), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
POST	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found).
DELETE	404 (Not Found), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

Resource Naming

In addition to utilizing the HTTP verbs appropriately, resource naming is arguably the most debated and most important concept to grasp when creating an understandable, easily leveraged Web service API. When resources are named well, an API is intuitive and easy to use. Done poorly, that same API can feel klutzy and be difficult to use and understand. Below are a few tips to get you going when creating the resource URIs for your new API.

Essentially, a RESTful API ends up being simply a collection of URIs, HTTP calls to those URIs, and some JSON and/or XML representations of resources, many of which will contain relational links. The RESTful principal of *addressability* is covered by the URIs. Each resource has its own address or URI—every interesting piece of information the server can provide is exposed as a resource. The constraint of *uniform interface* is partially addressed by the combination of URIs and HTTP verbs, and using them in line with the standards and conventions.

In deciding what resources are within your system, name them as nouns as opposed to verbs or actions. In other words, a RESTful URI should refer to a resource that is a *thing* instead of referring to an action. Nouns have properties but verbs do not, which is another way to distinguish them.

Some example resources are:

- Users of the system.
- Courses in which a student is enrolled.
- A user's timeline of posts.
- The users that follow another user.
- An article about horseback riding.

Each resource in a service suite will have at least one URI identifying it. It's best when that URI makes sense and adequately describes the resource. URIs should follow a predictable, hierarchical structure to enhance understandability and, therefore, usability: predictable in the sense that they're consistent, and hierarchical in the sense that data has structure and relationships. This is not a REST rule or constraint, but it enhances the API.

RESTful APIs are written for consumers. The name and structure of URIs should convey meaning to those consumers. It's often difficult to know what the data boundaries should be, but with understanding of your data, you most-likely are equipped to take a stab and what makes sense to return

as a representation to your clients. Design for your clients, not for your data.

Let's say we're describing an order system with customers, orders, line items, products, etc. Consider the URIs involved in describing the resources in this service suite:

Resource URI Examples

To insert (create) a new customer in the system, we might use:
POST http://www.example.com/customers

To read a customer with Customer ID# 33245:
GET http://www.example.com/customers/33245
The same URI would be used for PUT and DELETE, to update and delete, respectively.

Here are proposed URIs for products:
POST http://www.example.com/products
for creating a new product.

GET|PUT|DELETE http://www.example.com/products/66432
for reading, updating, or deleting product 66432, respectively.

Now, here is where it gets fun... What about creating a new order for a customer?
One option might be:
POST http://www.example.com/orders
That could work to create an order, but it's arguably outside the context of a customer.

Because we want to create an order for a customer (note the relationship), this URI perhaps is not as intuitive as it could be. It could be argued that the following URI would offer better clarity:
POST http://www.example.com/customers/33245/orders
Now we know we're creating an order for customer ID# 33245.

Now what would the following return?
GET http://www.example.com/customers/33245/orders
It would probably return a list of orders that customer #33245 has created or owns. Note: we may choose to not support DELETE or PUT for that URI since it's operating on a collection.

Now, to continue the hierarchical concept, what about the following URI?
POST http://www.example.com/customers/33245/orders/8769/lineitems
That might add a line item to order #8769 (which is for customer #33245). Right! GET for that URI might return all the line items for that order. However, if line items don't make sense only in customer context or do make sense outside the context of a customer, we would offer a *POST www.example.com/orders/8769/lineitems* URI.

Along those lines, because there may be multiple URIs for a given resource, we might also offer a *GET http://www.example.com/orders/8769* URI that supports retrieving an order by number without having to know the customer number.

To go one layer deeper in the hierarchy:
GET http://www.example.com/customers/33245/orders/8769/lineitems/1
Might return only the first line item in that same order.

By now you can see how the hierarchy concept works. There aren't any hard and fast rules. Just make sure the imposed structure makes sense to consumers of your services. As with everything in the craft of Software Development, naming is critical to success.

Look at some widely used APIs to get the hang of this and leverage the intuition of your teammates to refine your API resource URIs. Some example APIs are:

- Twitter: https://dev.twitter.com/docs/api
- Facebook: http://developers.facebook.com/docs/reference/api/
- LinkedIn: https://developer.linkedin.com/apis

Resource Naming Anti-Patterns

While we've discussed some examples of appropriate resource names, sometimes it's informative to see some anti-patterns. Below are some examples of poor RESTful resource URIs seen in the "wild." These are examples of what *not* to do!

First up, often services use a single URI to specify the service interface, using query-string parameters to specify the requested operation and/or HTTP verb. For example to update customer with ID 12345, the request for a JSON body might be:

GET http://api.example.com/services?op=update_customer&id=12345&format=json

By now, you're above doing this. Even though the 'services' URL node is a noun, this URL is not self-descriptive since the URI hierarchy is the same for all requests. Plus, it uses GET as the HTTP verb even though it's performing an update. This is counter-intuitive and is painful (even dangerous) to use as a client.

Here's another example following the same operation of updating a customer:

GET http://api.example.com/update_customer/12345

And its evil twin:

GET http://api.example.com/customers/12345/update

You'll see this one a lot as you visit other developer's service suites. Note that the developer is attempting to create RESTful resource names and has made some progress. But you're better than this— you're able to identify the verb phrase in the URL. Notice that we don't need to use the 'update' verb phrase in the URL because we can rely on the HTTP verb to inform that operation. Just to clarify, the following resource URL is redundant:

PUT http://api.example.com/customers/12345/update

With both PUT and 'update' in the request, we're offering to confuse our service consumers! Is 'update' the resource? So, we've spent some time beating the horse at this point. I'm certain you understand...

Pluralization

Let's talk about the debate between the pluralizers and the "singularizers"... Haven't heard of that debate? It *does* exist. Essentially, it boils down to this question of whether URI nodes in a hierarchy are named using singular or plural nouns? For example, should your URI for retrieving a representation of a customer resource look like this:

*GET http://www.example.com/**customer**/33245*

or:

*GET http://www.example.com/**customers**/33245*

There are good arguments on both sides, but the commonly-accepted practice is to *always use plurals*

in node names to keep your API URIs consistent across all HTTP methods. The reasoning is based on the concept that customers are a collection within the service suite and the ID (e.g. 33245) refers to one of those customers in the collection.

Using this rule, an example multi-node URI using pluralization would look like (emphasis added):

*GET http://www.example.com/**customers**/33245/**orders**/8769/**lineitems**/1*

with 'customers', 'orders', and 'lineitems' URI nodes all being their plural forms.

This implies that you only really need two base URLs for each root resource. One for creation of the resource within a collection and the second for reading, updating, and deleting the resource by its identifier. For example the creation case, using customers as the example, is handled by the following URL:

POST http://www.example.com/customers

And the read, update, and delete cases are handled by the following:

GET|PUT|DELETE http://www.example.com/customers/{id}

As mentioned earlier, there may be multiple URIs for a given resource, but as a minimum full CRUD capabilities are aptly handled with two simple URIs.

You ask if there is a case where pluralization doesn't make sense. Well, yes, in fact there is — when there isn't a collection concept in play. In other words, it's acceptable to use a singularized resource name when there can only be one of the resource—it's a singleton resource. For example, if there was a single, overarching configuration resource, you might use a singularized noun to represent that:

GET|PUT|DELETE http://www.example.com/configuration

Note the lack of a configuration ID and usage of POST verb. And say that there was only one configuration per customer, then the URL might be:

GET|PUT|DELETE http://www.example.com/customers/12345/configuration

Again, there's no ID for the configuration and no POST verb usage. Although, I'm sure that in both of these cases POST usage might be argued to be valid. Well... OK.

Returning Representations

As mentioned earlier, it is desirable for a service to support multiple representations of resources, including JSON and XML, as well as wrapped JSON and XML. I recommend JSON as the default representation, but services should allow clients to request alternative representations.

For a client to request a representation format, there is a question around whether to use the Accept header, a file-extension-style format specifier, query-string parameter, etc. Optimally, services would support all of those methods. However, the industry is currently converging on using a format specifier, which looks more like a file extension. Therefore, the recommendation is that, at a minimum, services support the use of file extensions such as '.json', '.xml' and wrapped file extensions such as '.wjson' and '.wxml'.

Using this technique, the representation format is specified in the URI, enhancing visibility. For example, *GET http://www.example.com/customers.xml* would return the list of customer representations in XML format. Likewise, *GET http://www.example.com/customers.json* would return a JSON representation. This makes the services simple to use from even the most basic client (such as 'curl') and is recommended.

Also, services should return the default representation format (presumably JSON) when a format specifier is not included in the URI. For example:

GET http://www.example.com/customers/12345
GET http://www.example.com/customers/12345.json

Both of the above return the 12345 customer resource in a JSON representation, which is the default format for this service.

GET http://www.example.com/customers/12345.xml

returns the 12345 customer resource in an XML representation, if supported. If an XML representation of this resource is not supported by this service, an HTTP 404 error should be returned.

Use of the HTTP Accept header is considered by many to be a more elegant approach, and is in keeping with the meaning and intent of the HTTP specification with regards to how clients notify HTTP servers of which content types they support. However, to support both wrapped and unwrapped responses from your services, in order to utilize the Accept header you must implement your own custom types—since there are no standard types for these formats. This increases the complexity of both clients and services greatly. See Section 14.1 of RFC 2616 (http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.1) for details on the Accept header. Supporting file-extension-style format specifiers is simple, straight-forward, gets the job done in the fewest number of characters, and easily supports scripting—without having to leverage HTTP headers.

In general, when we talk about REST services, XML is largely irrelevant. Barely anyone uses XML with REST, although supporting XML is recommended. XML standards and conventions are really not in play. In particular, namespaces are not used in a RESTful service context, nor should they be. They just muddy the waters and make things more complicated. So the XML that is returned is more JSON like—simple and easy to read, without the schema and namespace constraints—non-standard in other words, but parse-able.

Resource Discoverability Through Links (HATEOAS cont'd)

One of the guiding principals of REST (via the Uniform Interface constraint) is that application state is communicated via hypertext. This is often referred to as Hypertext As The Engine of Application State (HATEOAS), as mentioned above in the *What is Rest?* Section.

According to Roy Fielding's blog (at http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven), the most important part of a REST interface is its usage of hypertext. Further, he states that an API should be usable and understandable given an initial URI without prior knowledge or out-of-band information. That is, an API should be navigable via its links to various components of the data. Returning only data representations is discouraged.

This practice is not often followed by current industry leaders in services, reflecting that HATEOAS usage is higher on the maturity model. Looking around at many services, the convention is to return more data and fewer (or no) links. This is contrary to Fielding's REST constraints. Fielding says, “Every addressable unit of information carries an address... Query results are represented by a list of links with summary information, not by arrays of object representations.”

On the other hand, simply returning collections of links can be a major cause of network chattiness. In the real world, depending on requirements or use cases, chattiness of the API interface is managed by balancing how much “summary” data is included along with the relational hypertext links in service responses.

Also, full use of HATEOAS can increase implementation complexity and impose a significant burden on service clients, decreasing developer productivity on both client and server ends of the equation. Consequently, it is imperative to balance hyperlinking service implementations with available development resources.

A minimal set of hyperlinking practices provides major gains in service usability, navigability, and understandability while minimizing development impact and reducing the coupling between client and server. These minimal recommendations are for resources created via POST and for collections returned from GET requests, with additional recommendations for pagination cases, which are described below.

Minimal Linking Recommendations

In create use cases, the URI (link) for the newly-created resource should be returned in the *Location* response header and the response body should be empty—or contain only the ID of the newly-created resource.

For collections of representations being returned from a service, each representation should minimally carry a 'self' link property in its own *links* collection. Other links may be present in the returned representation as a separate link collection to facilitate pagination, with 'first', 'previous', 'next', and 'last' links where applicable.

See the examples in the *Link Format* section below for more information.

Link Format

Regarding overall link format standards, it is recommended to adhere to some semblance of the Atom, AtomPub, or Xlink style. JSON-LD is getting some traction too, but is not widely adopted yet (if it ever will be). Most widespread in the industry is usage of the Atom link style with a “rel” element and an “href” element that contains the full URI for the resource without any authentication or query-string parameters. The “rel” element can contain the standard values "alternate", "related", "self", "enclosure", and "via", plus “first”, “last”, “previous”, and “next” for pagination links. Use them where they make sense and add your own when needed.

Some of the XML Atom format concepts are somewhat irrelevant for links being represented in JSON. For instance, the METHOD property is not needed for a RESTful resource since the URIs are the same for a given resource, with all of the HTTP methods being supported (for CRUD behavior) — so listing them individually is overkill.

Let's make all this talk a little more concrete with some examples. Here's what the response would look like after creating a new resource with a call to

POST http://api.example.com/users

And here's an example set of response headers with the *Location* header set containing the new resource URI:

HTTP/1.1 201 CREATED
Status: 201
Connection: close
Content-Type: application/json; charset=utf-8
Location: http://api.example.com/users/12346

The body is either empty, or contains a wrapped response (see *Wrapped Responses* below).

Here is an example JSON response to a GET request that returns a collection of representations without pagination involved:

*{"data":[{"user_id":"42", "name":"Bob", **"links"**:[{"rel":"self", "href":"http://api.example.com/users/42"}]}, {"user_id":"22", "name":"Frank", **"links"**:[{"rel":"self", "href":"http://api.example.com/users/22"}]}, {"user_id":"125", "name": "Sally", **"links"**:[{"rel":"self", "href":"http://api.example.com/users/125"}]}]}*

Note the links array containing a single reference to "self" for each item in the collection. This array could potentially contain other relationships, such as children, parent, etc.

The final example is a JSON response to a GET request that returns a collection where pagination is involved (we're using three items per page), and we're on the third page of the collection:

*{"data":[{"user_id":"42", "name":"Bob", "links":[{"rel":"self", "href":"http://api.example.com/users/42"}]}, {"user_id":"22", "name":"Frank", "links":[{"rel":"self", "href":"http://api.example.com/users/22"}]}, {"user_id":"125", "name": "Sally", "links":[{"rel":"self", "href":"http://api.example.com/users/125"}]}], **"links"**:[{"rel":"first", "href":"http://api.example.com/users?offset=0&limit=3"}, {"rel":"last", "href":"http://api.example.com/users?offset=55&limit=3"}, {"rel":"previous", "href":"http://api.example.com/users?offset=3&limit=3"}, {"rel":"next", "href":"http://api.example.com/users?offset=9&limit=3"}]}*

In this example, the links collection in the response is populated for pagination purposes along with the link to "self" in each of the items in the collection. There could be additional links here related to the collection but not related to pagination. The simple summary is there are two places to include links in a collection. For each item in the collection (those in the *data* object, which is the collection of representations requested), include a links collection that, minimally, would contain a "self" reference. Then, in a separate object, *links*, include links that apply to the entire collection as applicable, such as pagination-related links.

For the create use case—create via POST, include a *Location* header with a link to the newly-created object.

Wrapped Responses

Services have the opportunity to return both HTTP status codes along with a body in the response. In many JavaScript frameworks, HTTP status response codes are not returned to the end-developer, often preventing the client from determining behavior based on that status code. Additionally, with the myriad response codes in the HTTP spec, often there are only a few that clients care about—frequently boiling down to 'success', 'error', or 'failure'. Consequently, it is beneficial to wrap responses in a representation that contains information about the response as well as the response itself.

One such proposal is that from OmniTI Labs, the so-called JSEND response. More information can be found at http://labs.omniti.com/labs/jsend. Another option is proposed by Douglas Crockford and can be found at http://www.json.org/JSONRequest.html.

In practice neither of these proposals adequately covers all cases. Basically, current best practice is to wrap regular (non-JSONP) responses with the following properties:

- **code** – contains the HTTP response status code as an integer.
- **status** – contains the text "success", "fail", or "error" where "fail" is for HTTP status response values from 500-599, "error" is for statuses 400-499, and "success" is for everything else (e.g. 1XX,

2XX and 3XX responses).

- **message** – only used for “fail” and “error” statuses to contain the error message. For internationalization (i18n) purposes, this could contain a message number or code, either alone or contained within delimiters.

- **data** – that contains the response body. In the case of “error” or “fail” statuses, this contains the cause, or exception name.

A successful response in wrapped style looks similar to this:
{"code":200,"status":"success","data":{"lacksTOS":false,"invalidCredentials":false,"authToken":"4ee683baa2a3332c3c86026d"}}

An example error response in wrapped style looks like this:
{"code":401,"status":"error","message":"token is invalid","data":"UnauthorizedException"}

In XML, these two wrapped responses would correspond to:

```
<response>
  <code>200</code>
  <status>success</status>
  <data class="AuthenticationResult">
    <lacksTOS>false</lacksTOS>
    <invalidCredentials>false</invalidCredentials>
    <authToken>1.0|idm|idm|4ee683baa2a3332c3c86026d</authToken>
  </data>
</response>
```

And:

```
<response>
  <code>401</code>
  <status>error</status>
  <message>token is invalid</message>
  <data class="string">UnauthorizedException</data>
</response>
```

Handling Cross-Domain Issues

We've all heard about working around the browser's same origin policy or common-source requirement. In other words, the browser can only make requests to the site it's currently displaying. For example, if the site currently being displayed is *www.Example1.com*, then that site cannot perform a request against *www.Example2.com*. Obviously, this impacts how sites access services.

Presently, there are two widely-accepted methods to support cross-domain requests: JSONP and Cross-Origin Resource Sharing (CORS). JSONP or "JSON with padding" is a usage pattern that provides a method to request data from a server in a different domain. It works by the service returning arbitrary JavaScript code instead of JSON. These responses are evaluated by the JavaScript interpreter, not parsed by a JSON parser. CORS, on the other hand, is a web browser technology specification, which defines ways for a web server to allow its resources to be accessed by a web page from a different domain. It is seen as a modern alternative to JSONP and is supported by all modern browsers. Therefore, JSONP is not recommended. Choose CORS whenever and wherever possible.

Supporting CORS

Implementing CORS on a server is as simple as sending an additional HTTP header in the response, for example:

Access-Control-Allow-Origin: *

An access origin of '*' should only be set if the data is meant for **public consumption**. In most cases the Access-Control-Allow-Origin header should specify **which domains** should be able to initiate a CORS request. Only URLs that need to be accessed cross-domain should have the CORS header set.

Access-Control-Allow-Origin: http://example.com:8080 http://foo.example.com

Allow only trusted domains in Access-Control-Allow-Origin header.

Access-Control-Allow-Credentials: true

Use this header only when necessary as it will send the cookies/sessions if the user is logged into the application.

These headers can be configured via the Web server, proxy or sent from the service itself. Implementing it within the services is not recommended as it's not flexible. Instead, use the second form, a space delimited list of appropriate domains configured on your Web server. More about CORS can be found at: http://enable-cors.org/.

Supporting JSONP

JSONP gets around the browser limitation by utilizing GET requests to perform all service calls. In essence, the requester adds a query-string parameter (e.g. jsonp="jsonp_callback") to the request, where the value of the "jsonp" parameter is the name of a javascript function that will be called when the response is returned.

There are severe limitations to the functionality enabled by JSONP since GET requests do not contain a request body and, therefore, information must be passed via query-string parameters. Also, to support PUT, POST, and DELETE operations, the effective HTTP method must also be passed as a query-string argument, such as _method=POST. Tunneling the HTTP method like this is not recommended and can open services up to security risks.

JSONP works on legacy browsers which preclude CORS support, but affects how services are built if they're going to support JSONP. Alternatively, JSONP can be implemented via a proxy. Overall, JSONP is being de-emphasized in favor of CORS. Favor CORS whenever possible.

To support JSONP on the server side, when the JSONP query-string parameter is passed in, the response must be manipulated as follows:

1. The response body must be wrapped as the parameter to the given javascript function in the jsonp parameter (e.g. jsonp_callback("<JSON response body>")).

2. Always return HTTP status 200 (OK) and return the actual status as part of the JSON response.

Additionally, it's also often necessary to include headers as part of the response body. This enables the JSONP callback method to make decisions on response handling based on the response body since it's not privy to the information in response headers and status.

An example error response following the above wrapped response recommendations is as follows (note: HTTP response status is 200):

jsonp_callback("{'code':'404', 'status':'error','headers':[],'message':'resource XYZ not found','data':'NotFoundException'}")

A successful creation response looks like this (still with an HTTP response status of 200):

jsonp_callback("{'code':'201', 'status':'error','headers':[{'Location':'http://www.example.com/customers/12345'}],'data':'12345'}")

Querying, Filtering, and Pagination

For large data sets, limiting the amount of data returned is important from a band-width standpoint. But it's also important from a UI processing standpoint as a UI often can only display a small portion of a huge data set. In cases where the dataset grows indefinitely, it's helpful to limit the amount of data returned by default. For instance, in the case of Twitter returning a person's tweets (via their home timeline), it returns up to 20 items unless otherwise specified in the request, and even then will return a maximum of 200.

Aside from limiting the amount of data returned, we also need to consider how to "page" or scroll through that large data set if more than that first subset needs retrieval. This is referred to as pagination—creating "pages" of data, returning known sections of a larger list and being able to page "forward" and "backward" through that large data set. Additionally, we may want to specify the fields or properties of a resource to be included in the response, thereby limiting the amount of data that comes back and we eventually want to query for specific values and/or sort the returned data.

There are combinations of two primary ways to limit query results and perform pagination. First, the indexing scheme is either page-oriented or item-oriented. In other words, incoming requests will specify where to begin returning data with either a "page" number, specifying a number of items per page, or specifying a first and last item number directly (in a range) to return. In other words the two options are, "give me page 5 assuming 20 items per page" or "give me items 100 through 120."

Service providers are split on how this should work. However, some UI tools, such as the Dojo JSON Datastore object, chooses to mimic the HTTP specification's use of byte ranges. It's very helpful if your services support that right out of the box so no translation is necessary between your UI toolkit and back-end services.

The recommendations below support both the Dojo model for pagination, which is to specify the range of items being requested using the *Range* header, and utilization of query-string parameters. By supporting both, services are more flexible—usable from both advanced UI toolkits, like Dojo, as well as by simple, straight-forward links and anchor tags. It shouldn't add much complexity to the development effort to support both options. However, if your services don't support UI functionality directly, consider eliminating support for the *Range* header option.

It's important to note that querying, filtering, and pagination are not recommended for all services. This behavior is resource specific and should not be supported on all resources by default. Documentation for the services and resources should mention which end-points support these more complex capabilities.

Limiting Results

The "give me items 3 through 55" way of requesting data is more consistent with how the HTTP spec utilizes the *Range* header for bytes so we use that metaphor with the Range header. However, "starting with item 2 give me a maximum of 20 items" is easier for humans to read, formulate, and understand so we use that metaphor in supporting the query-string parameters.

As mentioned above, the recommendation is to support use of both the HTTP *Range* header plus query-string parameters, *offset* and *limit*, in our services to limit results in responses. Note that, given support for both options, the query-string parameters should override the *Range* header.

One of the first questions you're going to ask is, "Why are we supporting two metaphors with these similar functions as the numbers in the requests will never match? Isn't that confusing?" Um... Those are two questions. Well, to answer your question, it may be confusing. The thing is, we want to make things in the query-string especially clear, easily-understood, human readable, and easy to construct and parse. The Range header, however, is more machine-based with its usage dictated to us via the HTTP specification.

In short, the Range header items value must be parsed, which increases complexity, plus the client side has to perform some computation in order to construct the request. Using the individual limit and offset parameters is easily-understood and created, usually without much demand on the human element.

Limiting via the Range Header

When a request is made for a range of items using a HTTP header instead of query-string parameters, include a *Range* header specifying the range as follows:

Range: items=0-24

Note that items are zero-based to be consistent with the HTTP specification in how it uses the *Range* header to request bytes. In other words, the first item in the dataset would be requested by a beginning range specifier of zero (0). The above request would return the first 25 items, assuming there were at least 25 items in the data set.

On the server side, inspect the *Range* header in the request to know which items to return. Once a *Range* header is determined to exist, it can be simply parsed using a regular expression (e.g. "items=(\\d+)-(\\d+)") to retrieve the individual range values.

Limiting via Query-String Parameters

For the query-string alternative to the Range header, use parameter names of *offset* and *limit*, where *offset* is the beginning item number (matches the first digit in the *items* string for the *Range* header above) and *limit* is the maximum number of items to return. A request using query-string parameters that matches the example in the Range Header section above is:

*GET http://api.example.com/resources?**offset=0&limit=25***

The *offset* value is zero-based, just like the *items* in the *Range* header. The value for *limit* is the maximum number of items to return. Services can impose their own default and maximum values for limit for when it's not specified in the query string. But please document those "invisible" settings.

Note that when the query-string parameters are used, the values should override those provided in the *Range* header.

Range-Based Responses

For a range-based request, whether via *Range* HTTP header or query-string parameters, the server should respond with a *Content-Range* header to indicate how many items are being returned and how many total items exist yet to be retrieved:

Content-Range: items 0-24/66

Note that the total items available (e.g. 66 in this case) is not zero-based. Hence, requesting the last few items in this data set would return a *Content-Range* header as follows:

Content-Range: items 40-65/66

According to the HTTP specification, it is also valid to replace the total items available (66 in this case) with an asterisk ("*") if the number of items is unknown at response time, or if the calculation of that number is too expensive. In this case the response header would look like this:

*Content-Range: items 40-65/**

However, note that Dojo or other UI tools may not support this notation.

Pagination

The above response-limiting schemes work for pagination by allowing requesters to specify the items within a dataset in which they're interested. Using the above example where 66 total items are available, retrieving the second "page" of data using a page size of 25 would use a *Range* header as follows:

Range: items=25-49

Via query-string parameters, this would be equivalent to:

*GET ...?**offset=25&limit=25***

Whereupon, the server (given our example) would return the data, along with a *Content-Range* header as follows:

Content-Range: 25-49/66

This works great for most things. However, occasionally there are cases where item numbers don't translate directly to rows in the data set. Also, for an extremely active data set where new items are regularly added to the top of the list, apparent "paging issues" with what look like duplicates can occur.

Date-ordered data sets are a common case like a Twitter feed. While you can still page through the data using item numbers, sometimes it's more beneficial and understandable to use an "after" or "before" query-string parameter, optionally in conjunction with the *Range* header (or query-string parameters, *offset* and *limit*).

For example, to retrieve up to 20 remarks around a given timestamp:

GET http://www.example.com/remarks/home_timeline?after=<timestamp>
Range: items=0-19

GET http://www.example.com/remarks/home_timeline?before=<timestamp>
Range: items=0-19

Equivalently, using query-string parameters:

GET http://www.example.com/remarks/home_timeline?after=<timestamp>&offset=0&limit=20
GET http://www.example.com/remarks/home_timeline?before=<timestamp>&offset=0&limit=20

For timestamp formatting and handling in different cases, please see the *Date Handling* section below.

If a service returns a subset of data by default or a maximum number of arguments even when the requester does not set a Range header, have the server respond with a *Content-Range* header to communicate the limit to the client. For example, in the home_timeline example above, that service call may only ever return 20 items at a time whether the requester sets the Range header or not. In that

case, the server should always respond with content range header such as:

Content-Range: 0-19/4125

or *Content-Range: 0-19/**

Filtering and Sorting Results

Another consideration for affecting results is the act of filtering data and/or ordering it on the server, retrieving a subset of data in a specified order. These concepts work in conjunction with pagination and result-limiting. They utilize query-string parameters, *filter* and *sort* respectively, to do their magic.

Again, filtering and sorting are complex operations and don't need to be supported by default on all resources. Document those resources that offer filtering and sorting.

Filtering

In this case, filtering is defined as reducing the number of results returned by specifying some criteria that must be met on the data before it is returned. Filtering can get quite complex if services support a complete set of comparison operators and complex criteria matching. However, it is quite often acceptable to keep things sane by supporting a simple equality, 'starts-with' or contains comparison.

Before we get started discussing what goes in the filter query-string parameter, it's important to understand why a single parameter vs. multiple query-string parameters is used. Basically, it comes down to reducing the possibility of parameter name clashes. We're already embracing the use of *offset*, *limit*, and *sort* (see below) parameters. Then there's *jsonp* if you choose to support it, the *format* specifier, and possibly *after* and *before* parameters. And those are just the query-string parameters discussed in **this** document. The more parameters we use on the query-string the more possibilities we have to have name clashes or overlap. Using a single filter parameter minimizes that.

Plus, it's easier from the server-side to determine if filtering functionality is requested by simply checking for the presence of that single *filter* parameter. Also, as complexity of your querying requirements increases, this single parameter option provides more flexibility in the future—for creating your own fully-functional query syntax (see OData comments below or at http://www.odata.org).

By embracing a set of common, accepted delimiters, equality comparison can be implemented in a straight-forward fashion. Setting the value of the filter query-string parameter to a string using those delimiters creates a list of name/value pairs which can be parsed easily on the server-side and utilized to enhance database queries as needed. The delimiters that have worked as conventions are the vertical bar (“|”) to separate individual filter phrases and a double colon (“::”) to separate the names and values. This provides a unique-enough set of delimiters to support the majority of use cases and creates a user-readable query-string parameter. A simple example will serve to clarify the technique. Suppose we want to request users with the name “Todd” who live in Denver and have the title of “Grand Poobah”. The request URI, complete with a query-string, might look like this:

GET http://www.example.com/users?filter="name::todd|city::denver|title::grand poobah”

The delimiter of the double colon (“::”) separates the property name from the comparison value, enabling the comparison value to contain spaces—making it easier to parse the delimiter from the value on the server.

Note that the property names in the name/value pairs match the name of the properties that would be returned by the service in the payload.

Simple but effective. Case sensitivity is certainly up for debate on a case-by-case basis, but in general, filtering works best when case is ignored. You can also offer wild-cards as needed using the asterisk ("*") as the value portion of the name/value pair.

For queries that require more-than simple equality or wild-card comparisons, introduction of operators is necessary. In this case, the operators themselves should be part of the value and parsed on the server side, rather than part of the property name. When complex query-language-style functionality is needed, consider introducing query concept from the Open Data Protocol (OData) Filter System Query Option specification (see http://www.odata.org/documentation/uri-conventions#FilterSystemQueryOption).

Sorting

For our purposes, sorting is defined as determining the order in which items in a payload are returned from a service. In other words, the sort order of multiple items appear in a response payload.

Again, convention here says to do something simple. The recommended approach is to utilize a *sort* query-string parameter that contains a delimited set of property names. The recommended behavior is, for each property name, sort in ascending order, and for each property prefixed with a dash ("-") sort in descending order. Separate each property name with a vertical bar ("|"), which is consistent with the separation of the name/value pairs in filtering, above.

For example, if we want to retrieve users in order of their last name (ascending), first name (ascending), and hire date (descending), the request might look like this:

GET http://www.example.com/users?sort=last_name|first_name|-hire_date

Note that again the property names match the name of the properties that would be returned by the service in the payload. Additionally, because of its complexity, offer sorting on a case-by-case basis for only resources that need it. Small collections of resources can be ordered on the client, if needed.

Service Versioning

Straight-up, versioning is hard, arduous, difficult, fraught with heartache, and even pain and extreme sadness — let's just say it adds a lot of complexity to an API and possibly to the clients that access it. Consequently, be deliberate in your API design and make efforts to not need versioned representations.

Favor not versioning instead of using versioning as a crutch for poor API design. You'll hate yourself in the morning if you need to version your APIs at all, let alone frequently. Lean on the idea that with the advent of using JSON for representations, clients can tolerate new properties appearing in a response without breaking. But even that is laden with danger in certain cases, such as changing the meaning of an existing property with either contents or validation rules.

Inevitably there will come a time when an API requires a change to its returned or expected representation that will cause consumers to break and that breaking change must be avoided. Versioning your API is the way to avoid breaking your clients and consumers.

Support Versioning via Content Negotiation

Historically, versioning was accomplished via a version number in the URI itself, with clients indicating which version of a resource they desired directly in the URI they requested. In fact, many of the "big boys" such as Twitter, Yammer, Facebook, Google, etc. frequently utilize version numbers in

their URIs. Even API management tools such as WSO2 have **required** version numbers in the exposed URLs.

This technique flies in the face of the REST constraints as it doesn't embrace the built-in header system of the HTTP specification, nor does it support the idea that a new URI should be added only when a new resource or concept is introduced — not representation changes. Another argument against it is that resource URIs aren't meant to change over time. A resource is a resource.

The URI should be simply to identify the resource — not its 'shape'. Another concept must be used to specify the format of the response (representation). That "other concept" is a pair of HTTP headers: *Accept* and *Content-Type*. The *Accept* header allows clients to specify the media type (or types) of the response they desire or can support. The *Content-Type* header is used by both clients and servers to indicate the format of the request or response body, respectively.

For example, to retrieve a user in JSON format:

```
# Request
GET http://api.example.com/users/12345
Accept: application/json; version=1

# Response
HTTP/1.1 200 OK
Content-Type: application/json; version=1

{"id":"12345", "name":"Joe DiMaggio"}
```

Now, to retrieve version 2 of that same resource in JSON format:

```
# Request
GET http://api.example.com/users/12345
Accept: application/json; version=2

# Response
HTTP/1.1 200 OK
Content-Type: application/json; version=2

{"id":"12345", "firstName":"Joe", "lastName":"DiMaggio"}
```

Notice how the URI is the same for both versions as it identifies the resource, with the Accept header being used to indicate the format (and version in this case) of the desired response. Alternatively, if the client desired an XML formatted response, the Accept header would be set to 'application/xml' instead, with a version specified, if needed.

Since the Accept header can be set to allow multiple media types, in responding to the request a server will set the Content-Type header on the response to the type that best matches what was requested by the client. Please see http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html for more information.

For example:

```
# Request
GET http://api.example.com/users/12345
```

```
Accept: application/json; version=1, application/xml; version=1
```

The above request, assuming the server supports one or both of the requested types, will either be in JSON or XML format, depending on which the server favors. But whichever type the server chooses, it will be set on the Content-Type header in the response.

For example, the response from the server if it favors *application/xml* would be:

```
# Response
HTTP/1.1 200 OK
Content-Type: application/xml; version=1

<user>
    <id>12345</id>
    <name>Joe DiMaggio</name>
</user>
```

To illustrate the use of Content-Type when sending data to the server, here is an example of creating a new user using JSON format:

```
# Request
POST http://api.example.com/users
Content-Type: application/json; version=1

{“name”:”Marco Polo”}
```

Or, if version 2 was in play:

```
# Request
POST http://api.example.com/users
Content-Type: application/json; version=2

{“firstName”:”Marco”, “lastName”:”Polo”}
```

What version is returned when no version is specified?

Supplying a version in each request is optional. As HTTP content-negotiation follows a “best match” approach with content types, so should your APIs. Using this “best match” concept, when the consumer does not specify a version, the API should return the oldest supported version of the representation.

For example, to retrieve a user in JSON format:

```
# Request
GET http://api.example.com/users/12345
Accept: application/json

# Response
HTTP/1.1 200 OK
```

Content-Type: application/json; version=1

{“id”:”12345”, “name”:”Joe DiMaggio”}

Similarly, when POSTing data without a version to an endpoint that supports multiple versions, the same rules as above apply — the lowest/earliest supported version is expected in the body. To illustrate, here is an example of creating a new user on a multi-version endpoint using JSON format (it expects version 1):

 # Request
POST http://api.example.com/users
Content-Type: application/json

{“name”:”Marco Polo”}

 # Response
HTTP/1.1 201 OK
Content-Type: application/json; version=1
Location: http://api.example.com/users/12345

{“id”:”12345”, “name”:”Marco Polo”}

Unsupported Versions Requested

When an unsupported version number is requested, including a resource version that has gone through the API deprecation lifecycle, the API should return an error response with 406 (Not Acceptable) HTTP status code. In addition, the API should return a response body with Content-Type: application/json that contains a JSON array of supported content types for that endpoint.

 # Request
For example:
GET http://api.example.com/users/12345
Content-Type: application/json; version=**999**

 # Response
HTTP/1.1 406 NOT ACCEPTABLE
Content-Type: application/json

[“application/json; version=1”, “application/json; version=2”, “application/xml; version=1”, “application/xml; version=2”]

When Should I Create a New Version?

In API development there are many ways to break a contract and negatively impact your clients. If you are uncertain of the consequences of your change it is better to play it safe and consider versioning.

There are several factors to consider when you are trying to decide if a new version is appropriate or if a modification of an existing representation is sufficient and acceptable.

Changes that will break contracts

- Changing a property name (e.g.. “name” to “firstName”)
- Removal of a property
- Changing property data type (numeric to string, boolean to bit/numeric, string to datetime, etc.)
- Validation rule change
- In Atom style links, modifying the “rel” value.
- A required resource being introduced into an existing workflow
- Resource concept/intent change; the concept/intent or the meaning of the resource’s state has a different meaning from its original. Examples:
    - A resource with the content type text/html once meant that the representation would be a collection of “links” to all supported media types, new text/html representation means “web browser form” for user input
    - An API populating an “endTime” on the resource “.../users/{id}/exams/{id}” once meant the student submitted the exam at that time, the new meaning is that it will be the scheduled end time of the exam.
- Adding new fields that came from an existing resource with the intent of deprecating the existing resource. Combining two resources into one and deprecating the two original resources.
    - There are two resources, “.../users/{id}/dropboxBaskets/{id}/messages/{id}” and “.../users/{id}/dropboxBaskets/{id}/messages/{id}/readStatus”. The new requirement is to put the properties from the readStatus resource into the individual message resource and deprecate the readStatus resource. This will cause the removal of a link to the readStatus resource in the individual messages resource.

While this list is not full-inclusive, it gives you an idea of the types of changes that will cause havoc for your clients and require a new resource or a new version.

Changes considered non-breaking

- New properties added to a JSON response.
- New/additional “link” to other resources.
- New content-type supported formats.
- New content-language supported formats.
- Casing is irrelevant as both the API producer and consumer should handle varied casing.

At What Level Should Versioning Occur?

It is recommended to version at the individual resource level. Some changes to an API such as modifying the workflow may require versioning across multiple resource to prevent breaking clients.

Use Content-Location to Enhance Responses

Optional. See RDF spec.

Links with Content-Type

Atom-style links support a 'type' property. Provide enough information so that clients can construct necessary calls to specific version & content type.

Finding Out What Versions are Supported

How many versions should I support at once?

Since maintaining many versions becomes cumbersome, complex, error prone, and costly you should support no more than 2 versions for any given resource.

Deprecated

The term deprecated is intended to be used to communicate that a resource is still available in the API, but will become unavailable and no longer exist in the future. *Note: The length of time in deprecation will be determined by the deprecation policy- not yet defined.*

How do I inform clients about deprecated resources?

Many clients will be using resources that are to be deprecated after new versions are introduced and in doing so, they will need ways to discover and monitor their applications use of deprecated resources. When a deprecated resource is requested, the API should return a normal response with the Pearson custom Header “Deprecated” in a boolean format. Below is an example.

```
# Request
GET http://api.example.com/users/12345
Accept: application/json
Content-Type: application/json; version=1


# Response
HTTP/1.1 200 OK
Content-Type: application/json; version=1
Deprecated: true

{“id”:”12345”, “name”:”Joe DiMaggio”}
```

Date/Time Handling

Dates and timestamps can be a real headache if not dealt with appropriately and consistently. Timezone issues can crop up easily and since dates are just strings in JSON payloads, parsing is a real issue if the format isn't known, consistent, or specified.

Internally, services should store, process, cache, etc. such timestamps in UTC or GMT time. This alleviates timezone issues with both dates and timestamps.

Date/Time Serialization In Body Content

There's an easy way around all of this—always use the same format, including the time portion (along with timezone information) in the string. ISO 8601 time point format is a good solution, using the fully-enhanced format that includes hours, minutes, seconds, and a decimal fraction of seconds (e.g. yyyy-MM-dd'T'HH:mm:ss.SSS'Z'). It is recommended that ISO 8601 be used for all dates represented in REST service body content (both requests and responses).

Incidentally, for those doing Java-based services, the DateAdapterJ library easily parses and formats ISO8601 dates, time points, and HTTP 1.1 header (RFC 1123) formats, with its DateAdapter, Iso8601TimepointAdapter, and HttpHeaderTimestampAdapter implementation classes, respectively. It can be downloaded at https://github.com/tfredrich/DateAdapterJ.

For those creating browser-based UIs, the ECMAScript 5 specification includes parsing and creating ISO8601 dates in JavaScript natively, so it should be making its way into all mainstream browsers as we speak. If you're supporting older browsers that don't natively parse those dates, a JavaScript library or fancy regular expression is in order. A couple of sample JavaScript libraries that can parse and produce ISO8601 Timepoints are:

http://momentjs.com/

http://www.datejs.com/

Date/Time Serialization In HTTP Headers

While the above recommendation works for JSON and XML content in the content of an HTTP request or response, the HTTP specification utilizes a different format for HTTP headers. Specified in RFC 822, which was updated by RFC 1123, that format includes various date, time, and date-time formats. However, it is recommended to always use a timestamp format, which ends up looking like this in your request headers:

Sun, 06 Nov 1994 08:49:37 GMT

Unfortunately, it doesn't account for a millisecond or decimal fraction of a second in its format. The Java SimpleDateFormat specifier string is: "EEE, dd MMM yyyy HH:mm:ss 'GMT'"

Securing Services

Authentication is the act of verifying that a given request is from someone (or some system) that is known to the service and that the requestor is who they say they are. While authentication is the act of verifying a requestor is who they say they are, authorization is verifying the requestor has permission to perform the requested operation.

Essentially, the process goes something like this:

1. Client makes a request, including an authentication token in a *X-Authorization* header or *token* query-string parameter in the request.

2. Service verifies presence of the authorization token, validates it (that it's valid and not expired), and parses or loads the authentication principal based on the token contents.

3. Service makes a call to the authorization service providing authentication principal, requested

resource, and required permission for operation.

4. If authorized, service continues with normal processing.

#3 above could be expensive, but assuming a cacheable access-control list (ACL), it is conceivable to create an authorization client that caches the most-recent ACLs to validate locally before making remote calls.

Authentication

Current best practice is to use OAuth for authentication. OAuth2 is highly recommended, but it's still in draft state. OAuth1 is definitely an acceptable alternative. 3-Legged OAuth is also an option for certain cases. Read more about the OAuth specification at http://oauth.net/documentation/spec/.

OpenID is an additional option. However, it is recommended that OpenID be used as an *additional* authentication option, leveraging OAuth as primary. Read more about the OpenID specification at http://openid.net/developers/specs/.

Transport Security

All authentication should use SSL. OAuth2 requires the authorization server and access token credentials to use TLS.

Switching between HTTP and HTTPS introduces security weaknesses and best practice is to use TLS by default for all communication.

Authorization

Authorization for services is not really any different than authorization for any application. It's based on the question, "Does this **principal** have the requested **permission** on the given **resource**?" Given that simple trifecta of data (principal, resource, and permission), it's fairly easy to construct an authorization service that supports the concepts. Using those generic concepts, it is possible to have a cacheable access control list (ACL) for each principal.

Application Security

The same principles in developing a secure web application hold true for RESTful services.

- Validate all input on the server. Accept "known" good input and reject bad input.
- Protect against SQL and NoSQL injection.
- Output encoded data using known libraries such as Microsoft's Anti-XSS or OWASP's AntiSammy.
- Restrict the message size to the exact length of the field.
- Services should only display generic error messages.
- Consider business logic attacks. For example, could an attacker skip through a multi-step ordering process and order a product without having to enter credit card information?
- Log suspicious activity.

RESTful Security Considerations:

- Validate JSON and XML for malformed data.
- Verbs should be restricted to the allowable methods. For example, a GET request should not be able to delete an entity. A GET would read the entity while a DELETE would remove the entity.

- Be aware of race conditions.

API gateways can be used to monitor, throttle, and control access to the API. The following can be done by a gateway or by the RESTful service.

- Monitor usage of the API and know what activity is good and what falls out of normal usage patterns.
- Throttle API usage so that a malicious user cannot take down an API endpoint (DOS attack) and have the ability to block a malicious IP address.
- Store API keys in a cryptographically secure keystore.

Caching and Scalability

Caching enhances scalability by enabling layers in the system to eliminate remote calls to retrieve requested data. Services enhance cache-ability by setting headers on responses. Unfortunately, caching-related headers in HTTP 1.0 are different than those in HTTP 1.1, so services should support both. Below is a table of minimal headers required to support caching for GET requests, along with a description of appropriate values.

HTTP Header	Description	Example
Date	Date and time the response was returned (in RFC1123 format).	Date: Sun, 06 Nov 1994 08:49:37 GMT
Cache-Control	The maximum number of seconds (max age) a response can be cached. However, if caching is not supported for the response, then no-cache is the value.	Cache-Control: 360 Cache-Control: no-cache
Expires	If max age is given, contains the timestamp (in RFC1123 format) for when the response expires, which is the value of Date (e.g. now) plus max age. If caching is not supported for the response, this header is not present.	Expires: Sun, 06 Nov 1994 08:49:37 GMT
Pragma	When Cache-Control is 'no-cache' this header is also set to 'no-cache'. Otherwise, it is not present.	Pragma: no-cache
Last-Modified	The timestamp when the resource itself was modified last (in RFC1123 format).	Last-Modified: Sun, 06 Nov 1994 08:49:37 GMT

To simplify, here's an example header set in response to a simple GET request on a resource that enables caching for one day (24 hours):

Cache-Control: 86400
Date: Wed, 29 Feb 2012 23:01:10 GMT
Last-Modified: Mon, 28 Feb 2011 13:10:14 GMT
Expires: Thu, 01 Mar 2012 23:01:10 GMT

And below is an example of a similar response that disables caching altogether:

Cache-Control: no-cache
Pragma: no-cache

The ETag Header

The ETag header is useful for validating the freshness of cached representations, as well as helping with conditional read and update operations (GET and PUT, respectively). Its value is an arbitrary string for the version of a representation. However, it also should be different for each format of a representation—the ETag for a JSON response will be different for the same resource represented in XML. The value for the ETag header can be as simple as a hash of the underlying domain object (e.g. Object.hashcode() in Java) with the format included in the hash. It is recommended to return an ETag header for each GET (read) operation. Additionally, make sure to surround the ETag value in double quotes. For example:

ETag: "686897696a7c876b7e"

HTTP Status Codes (Top 10)

Below are the most commonly-used HTTP status codes returned from RESTful services or APIs along with a brief summary of their commonly-accepted usage. Other HTTP status codes are used occasionally, but are either specializations or more advanced. Most service suites are well served by supporting only these, or even a sub-set.

200 (OK) – General success status code. Most common code to indicate success.

201 (CREATED) – Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource. Response body content may or may not be present.

204 (NO CONTENT) – Status when wrapped responses are not used and nothing is in the body (e.g. DELETE).

304 (NOT MODIFIED) – Used in response to conditional GET calls to reduce band-width usage. If used, must set the Date, Content-Location, and Etag headers to what they would have been on a regular GET call. There must be no response body.

400 (BAD REQUEST) – General error when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

401 (UNAUTHORIZED) – Error code for a missing or invalid authentication token.

403 (FORBIDDEN) – Error code for user not authorized to perform the operation, doesn't have rights to access the resource, or the resource is unavailable for some reason (e.g. time constraints, etc.).

404 (NOT FOUND) – Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

409 (CONFLICT) – Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, deleting root objects when cascade-delete not supported are a couple of examples.

500 (INTERNAL SERVER ERROR) – The general catch-all error when the server-side throws an exception.