RESTful Service Best Practices

Recommendations for Creating Web Services

Todd Fredrich Pearson eCollege toddf@ecollege.com



Table of Contents

| Introduction | 5 |
|---|----|
| What is REST? | 5 |
| Uniform Interface | 6 |
| Resource-Based | 6 |
| Manipulation of Resources Through Representations | 6 |
| Self-descriptive Messages | 6 |
| Hypermedia as the Engine of Application State (HATEOAS) | 6 |
| Stateless | 6 |
| Cacheable | 7 |
| Client-server | 7 |
| Layered system | |
| Code on demand (optional) | 7 |
| REST vs. SOAP | 8 |
| REST Quick Tips | 8 |
| Not SOAP | 8 |
| Use HTTP Verbs to Mean Something. | |
| Sensible Resource Names | |
| XML and JSON | |
| Use Abstraction Where it's Helpful | |
| HTTP Verbs | 9 |
| GET | |
| PUT | |
| POST | |
| PUT vs POST for Creation | 11 |
| DELETE | |
| Resource Naming | |
| Resource URI Examples | |
| Pluralization | 14 |
| Returning Representations | |
| Resource Discoverability Through Links (HATEOAS cont'd) | |
| Wrapped Responses | |
| Handling JSONP | |
| Limiting Results, Pagination and Filtering. | 17 |
| Limiting Results | |
| Pagination | |
| Filtering and Sorting Results | |
| Filtering | |
| Sorting | |
| Service Versioning. | |
| Date/Time Handling | |
| Securing Services | |
| Authentication | 22 |

| RESTRIC Service Dest Fractices | 02/10/12 |
|--------------------------------|----------|
| Authorization | 22 |
| Monitoring and Instrumentation | |
| Monitoring | |
| Instrumentation | 22 |
| Caching and Scalability | 23 |
| HTTP Status Codes (Top 10) | 24 |
| Additional Resources | 25 |
| Books | 25 |
| Websites | 25 |
| | |

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 the 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 principals 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 visa 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 the 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 discussion around 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. These constraints, applied to the architecture, were originally communicated by Roy Fielding in his doctoral dissertation (see http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) and defines 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 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 metadata attached, it 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 objects or related objects.

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

Stateless

As REST is an acronym for **RE**presentational State Transfer, *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 it's processing, the appropriate state, or the piece(s) of state that matter, are communicated back to the client via headers, status and response body.

Most of us who have been in the industry for a while are accustomed to programming within a container which provides us with 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 for 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 certain a 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 vs. SOAP

If you've read this far, there's a good chance you are already on-board with creation of RESTful Web services and don't need convincing that SOAP is "so 2001." On the other hand, it's possible you've been told that REST services are the "best thing since sliced bread—learn them..." and maybe you need a bit of a conversion experience. We don't want to beleaguer the point here since this document is about RESTful best practices. Besides, we're not really going to change anyone's mind with words. But...

Why has SOAP taken a back seat to REST in recent years? In short, it's complexity: adoption of public-facing SOAP services is slow because of complexity in the payload—both with parsing the envelope and the interoperability issues caused by the marshaled data types. Plus, complexity of learning WSDL, versioning, and the understandability of each interface has helped to slow adoption.

Usage of REST-style services with their common interface is much easier. Also, the interoperability concerns largely disappear since JSON is normally the serialization medium—it's just a string and it doesn't support any fancy types (like dates, etc.). Versioning is simplified too, because marshaling to/from JSON tends to not care about new data elements in the payload—they are ignored—and API version numbers are simply in the request (more in this later, in Versioning Services).

To summarize, it's much simpler to utilize a REST-style service, whether it's from a Web UI via JavaScript, or server-side code in various languages such as Ruby, Java, C#, Python, etc. And while we don't want to always do things "because everyone's doing it", in this case it doesn't hurt that the biggest, most respected Internet companies such as Amazon, Google, LinkedIn Twitter and FaceBook are utilizing REST service-style architectures to accomplish extreme scale.

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:

Not SOAP

There are legitimate reasons to have a SOAP API, and translating a massive API from a legacy SOAP to REST interface is quite difficult, but it's incredibly difficult to consume SOAP if you have to develop a client library from scratch, access it via a Web UI, or do anything non-standard.

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. It makes things more clear. Use the HTTP methods to specify the verb portion of the request.

XML and JSON

Unless the costs of offering both JSON and XML are staggering, offer them 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. And 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 use 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.

Use Abstraction Where it's Helpful

The API implementation does not have to mimic the underlying application or database architecture. If there's a common action that clients will use the API for, make that action easy, even if it means compromising on ideology. In other words, it's acceptable (even encouraged) to have the API support entire use cases instead of fine-grained (e.g. CRUD) services.

HTTP Verbs

The HTTP verbs comprise a major portion of our "uniform interface" constraint and provide us 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 methods, 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/bucket/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 it once has the same effect as calling it 10 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 not 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 in the case where 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/bucket/secret_stuff

On successful update, return 200 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 as 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 recommended to use POST for non-idempotent requests.

POST

The POST verb is most-often utilized for creation of 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, use PUT when the client is in charge of deciding which URI (via it's resource name or ID) the new resource will have: if the client knows what the resulting URI (or resource ID) will be, use PUT. 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/bucket/sample

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a JSEND-style 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 idempotency, 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.

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 as verbs do not, just another distinguishing factor.

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. And 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, hierarchical in the sense that data has structure—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, 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

And 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

Probably a list of orders that customer #33245 has created or owns. Note: we may choose to not support DELETE or PUT for that url 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 also 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, only 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

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...

Should URI nodes in your hierarchy be 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.

Returning 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, industry is currently converging on using a format specifier, which looks more like a file extension.

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.

As added support, if your service suites support both raw JSON and XML responses as well as wrapped responses, specifying which is only possible via a format specifier or query-string parameter since there is no standard content type to distinguish between wrapped JSON and plain, unwrapped JSON...

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, nor should they be use in a RESTful service context. It just muddies the waters and makes 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 respresentations is discouraged.

This practice is not often followed by current industry leaders in services. Looking around at many services, convention is to return more data and less (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."

Hyperlinking representations returned by services is imperative to improve its usability, navigability and understandability. For instance, in create use cases, the URI (link) for the newly-created resource should be returned in the *Location* response header and the response body be empty—or contain only the ID of the newly-created resource. Additionally, as mentioned above, every addressable unit of information should carry a link property. Other links may be present in the returned data also to facilitate pagination or other concepts.

Depending on requirements or use cases, chattiness of the interface is managed by balancing how much "summary" data is included along with the relational hypertext links.

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", "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.

Wrapped Responses

Services have the opportunity to return both HTTP status codes along with a body in the response. In

</response>

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 read about 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" which contains the HTTP response status code as an integer.

"status" which 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.

"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>
```

Handling JSONP

JSONP is a technique that works around the browser's common-source requirement. 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.

On the server side, if a jsonp query-string parameter is passed in, the response must be manipulated a bit 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'}")
```

Limiting Results, Pagination and Filtering

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.

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 specify 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 specifications 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 follow the Dojo model for pagination, which is to specify the range of items being requested using the *Range* header.

Limiting Results

The "give me items 3 through 55" way of requesting data is explicit and follows how the HTTP spec utilizes the Range header for bytes. It solves both pagination and results-limiting within a single range specifier. So it ends up being quite clear and understandable.

Therefore, the recommendation is to use the HTTP *Range* header to specify the items desired. When a request is made for a range of items, 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.

For a range-based request, 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 *Range* header scheme works 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:

RESTful Service Best Practices

Range: items=25-49

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

Content-Range: 25-49/66

This is 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.

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

http://www.example.com/remarks/home_timeline?after=<timestamp>

Range: items=0-19

<u>http://www.example.com/remarks/home_timeline?before</u> = < timestamp >

Range: items=0-19

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

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 comparison.

By embracing a set of common, accepted delimiters, equality comparison can be implemented in 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 Poomba". The request URI, complete with query-string might look like this:

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

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 ("*").

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 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 default 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.

Service Versioning

Services should be versioned as early as possible in the development cycle. Any initial public or internal release should be versioned right out of the gate.

As a developer, it's easiest to use an API when that API is accessible via simple tools like a browser, browser plug-ins or command-line tools like 'curl'. Therefore, in the case of versioning, the concept of 'visibility' comes very much into play. This means that it should be very obvious to the API consumer which version of that API they are consuming. Consequently, to enhance that visibility it is recommended to place a version number directly in resource URIs, very high in the URI node hierarchy.

This technique flies in the face of much academic REST conversations as it doesn't embrace the built-in

header system of the HTTP specification using etags, or that a new URI should be added only when a new concept is introduced. Furthermore, another argument against it is that resource URIs aren't meant to change over time—when something unrelated to the resource, like a new API version comes out.

However, putting the version in the URI makes the API easier to use, test, and verify that the appropriate resource representation version is being requested. It could also be argued that since returned values are actually representations—not the resource itself—the version number reflects a resource representation appropriately. Additionally, many of the "big boys" such as Twitter, Yammer, Facebook, Google, etc. frequently utilize version numbers in their URIs.

The current recommendation is to support versioning via version numbers directly in resource URIs. It makes the version visible and a versioned API easier to understand and use correctly.

Version numbers in URIs should be high in the node hierarchy, preferably as the first node, for example: api.example.com/v1/users or www.example.com/api/v1/users.

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.

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').

While the above recommendation works for JSON and XML content, the HTTP specification utilizes a different format. 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'"

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

Incidentally, for those doing Java-based services, the DateAdapterJ library easily parses and formats ISO8601 dates and 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.

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 authentication token in *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 a 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 is 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/.

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 tri-fecta of data (principal, resource, and permission), it's fairly easy to construct an authorization service that supports the concepts. Where Principal is the person or system who is granted a permission on a resource. Using those generic concepts, it is possible to have a cacheable access control list (ACL) for each principal.

Monitoring and Instrumentation

Monitoring

<coming soon>

Instrumentation

<coming soon>

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 |
| ETag | An arbitrary string for the version of a representation. Note that this should be different for JSON and XML representations. This could be as simple as a hash of the data with the format included as part of the hash. | ETag: 123456787654 |
| Last-Modified | The timestamp that the resource itself was modified last (in RFC1123 format). | Last-Modified: Sun, 06 Nov 1994 08:49:37 GMT |

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 JSEND-style responses are not used and nothing is in the body (e.g. DELETE).

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

403 (FORBIDDEN) – Error code for user not authorized to perform the operation 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.

Additional Resources

Books

REST API Design Rulebook, Mark Masse, 2011, O'Reilly Media, Inc.

RESTful Web Services, Leonard Richardson and Sam Ruby, 2008, O'Reilly Media, Inc.

RESTful Web Services Cookbook, Subbu Allamaraju, 2010, O'Reilly Media, Inc.

REST in Practice: Hypermedia and Systems Architecture, Jim Webber, et al., 2010, O'Reilly Media, Inc.

APIs: A Strategy Guide, Daniel Jacobson; Greg Brail; Dan Woods, 2011, O'Reilly Media, Inc.

Websites

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

http://www.json.org/

https://github.com/tfredrich/DateAdapterJ

http://openid.net/developers/specs/

http://oauth.net/documentation/spec/

http://www.json.org/JSONRequest.html

http://labs.omniti.com/labs/jsend

http://rov.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

https://developer.linkedin.com/apis

http://developers.facebook.com/docs/reference/api/

https://dev.twitter.com/docs/api