REST API

Building RESTful web services, like other programming skills is **part art, part science**. As the Internet industry progresses, creating a REST API becomes more concrete with emerging best practices. As RESTful web services don't follow a prescribed standard except for HTTP, it's important to build your RESTful API in accordance with industry best practices to ease development and increase client adoption.

Presently, there aren't a lot of REST API guides to help the lonely developer. RestApiTutorial.com is dedicated to tracking REST API best practices and making resources available to enable quick reference and self education for the development crafts-person. We'll discuss both the art and science of creating REST Web services.

—Todd Fredrich, *REST API Expert*

Jump in with What Is REST?, an overview of concepts and constraints of the RESTful architecture.

What Is REST?

Video

The REST architectural style describes six constraints. These constraints, applied to the architecture, were originally communicated by Roy Fielding in his doctoral dissertation (seehttp://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) and defines the basis of RESTful-style.

The six constraints are: (click the constraint to read more)

- 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 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 REpresentational 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 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 API Quick Tips

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

Use HTTP Verbs to Make Your Requests Mean Something

API consumers are capable of sending GET, POST, PUT, and DELETE verbs, which greatly enhance the clarity of a given request.

Generally, the four primary HTTP verbs are used as follows:

GET
: Read a specific resource (by an identifier) or a collection of resources.

PUT
: Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is know before-hand.

DELETE
: Remove/delete a specific resource by an identifier.

POST
: Create a new resource. Also a catch-all verb for operations that don't fit into the other categories.

Note

GET requests must not change any underlying resource data. Measurements and tracking which update data may still occur, but the resource data identified by the URI should not change.

Provide Sensible Resource Names

Producing a great API is 80% art and 20% science. Creating a URL hierarchy representing sensible resources is the art part. Having sensible resource names (which are just URL paths, such as /customers/12345/orders) improves the clarity of what a given request does.

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

Here are some quick-hit rules for URL path (resource name) design:

- Use identifiers in your URLs instead of in the query-string. Using URL query-string parameters is fantastic for filtering, but not for resource names.
    - **Good:** /users/12345
    - **Poor:** /api?type=user&id=23
- Leverage the hierarchical nature of the URL to imply structure.
- Design for your clients, not for your data.
- Resource names should be nouns. Avoid verbs as resource names, to improve clarity. Use the HTTP methods to specify the verb portion of the request.
- Use plurals in URL segments to keep your API URIs consistent across all HTTP methods, using the collection metaphor.

    - **Recommended:** /customers/33245/orders/8769/lineitems/1
    - **Not:** /customer/33245/order/8769/lineitem/1
- Avoid using collection verbiage in URLs. For example 'customer_list' as a resource. Use pluralization to indicate the collection metaphor (e.g. customers vs. customer_list).
- Use lower-case in URL segments, separating words with underscores ('_') or hyphens ('-'). Some servers ignore case so it's best to be clear.
- Keep URLs as short as possible, with as few segments as makes sense.

Use HTTP Response Codes to Indicate Status

Response status codes are part of the HTTP specification. There are quite a number of them to address the most common situations. In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes. For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201. A list of valid HTTP status codes is available here which lists detailed descriptions of each.

Suggested usages for the "Top 10" HTTP Response Status Codes are as follows:

200 OK
General success status code. This is the most common code. Used 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 (on POST). Response body content may or may not be present.
204 NO CONTENT
Indicates success but nothing is in the response body, often used for DELETE and PUT operations.
400 BAD REQUEST
General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.
401 UNAUTHORIZED
Error code response for missing or invalid authentication token.
403 FORBIDDEN
Error code for when the user is 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.
405 METHOD NOT ALLOWED
Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST */users/12345* where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"
409 CONFLICT
Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.
500 INTERNAL SERVER ERROR

Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

Offer Both JSON and XML

Favor JSON support unless you're in a highly-standardized and regulated industry that requires XML, schema validation and namespaces, and offer both JSON and XML unless the costs are staggering. Ideally, let consumers switch between formats using the HTTP Accept header, or by just changing an extension from .xml to .json on the URL.

Be aware that as soon as we start talking about XML support, we start talking about schemas for validation, namespaces, etc. Unless required by your industry, avoid supporting all that complexity initially, if ever. JSON is designed to be simple, terse and functional. Make your XML look like that if you can.

In other words, make the XML that is returned 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, the cost of XML will be staggering. In my experience no one has used XML responses anyway for the last several years, it's just too expensive to consume.

Note that JSON-Schema offers schema-style validation capabilities, if you need that sort of thing.

Create Fine-Grained Resources

When starting out, it's best to create APIs that mimic the underlying application domain or database architecture of your system. Eventually, you'll want aggregate services that utilize multiple underlying resources to reduce chattiness. However, 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 (search HATEOAS). While services are still useful without them, APIs become more self-descriptive and discoverable when links are returned in the response. At the very least, a 'self' link reference informs clients how the data was or can be retrieved. Additionally, utilize the HTTP Location header to contain a link on resource creation via POST (or PUT). For collections returned in a response that support pagination, 'first', 'last', 'next' and 'prev' links at a minimum are very helpful.

Regarding linking formats, there are many. The HTTP Web Linking Specification (RFC5988) explains a link as follows:

a link is a typed connection between two resources that are identified by Internationalised Resource Identifiers (IRIs) [RFC3987], and is comprised of:

- A context IRI,
- a link relation type
- a target IRI, and
- optionally, target attributes.

A link can be viewed as a statement of the form "{context IRI} has a {relation type} resource at {target IRI}, which has {target attributes}."

At the very least, place links in the HTTP Link header as recommended in the specification, or embrace a JSON representation of this HTTP link style (such as Atom-style links, see: RFC4287) in your JSON representations. Later, you can layer in more complex linking styles such as HAL+JSON, Siren, Collection+JSON, and/or JSON-LD, etc. as your REST APIs become more mature.

Using HTTP Methods for RESTful Services

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, PATCH, 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.

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

HTTP Verb	**CRUD**	**Entire Collection (e.g. /customers)**	**Specific Item (e.g. /customers/{id})**
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	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	Update/Replace	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.
PATCH	Update/Modify	404 (Not Found), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	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.

Below is a more-detailed discussion of the main HTTP methods. Click on a tab for more information about the desired HTTP method.

POST

The POST verb is most-often utilized to **create** 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.

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 nor 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.

Examples:

- *POST http://www.example.com/customers*
- *POST http://www.example.com/customers/12345/orders*

GET

The HTTP GET method is used to **read** (or retrieve) 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).

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 never modify any resources on the server.

Examples:

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

PUT

PUT is most-often utilized for **update** capabilities, PUT-ing to a known resource URI with the 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).

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.

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 strongly recommended to use POST for non-idempotent requests.

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*

PATCH

PATCH is used for **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.

This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.

PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag in an If-Match header on the PATCH request.

Examples:

- *PATCH http://www.example.com/customers/12345*
- *PATCH http://www.example.com/customers/12345/orders/98765*
- *PATCH http://www.example.com/buckets/secret_stuff*

DELETE

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

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 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, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.

Examples:

- *DELETE http://www.example.com/customers/12345*
- *DELETE http://www.example.com/customers/12345/orders*
- *DELETE http://www.example.com/bucket/sample*

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

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 as the URI hierarchy is the same for all requests. Plus, it uses GET as the HTTP verb even though we're 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 —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...

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.

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, 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.

What Is Idempotence?

Video

Idempotence

Idempotence is a funky word that often hooks people. Idempotence is sometimes a confusing concept, at least from the academic definition.

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. 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 (no 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, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

HTTP Status Codes

This page is created from HTTP status code information found at ietf.org and Wikipedia. Click on the **category heading** or the **status code** link to read more.

1xx Informational

100 Continue

The client SHOULD continue with its request. This interim response is used to inform the client that the initial part of the request has been received and has not yet been rejected by the server. The client SHOULD continue by sending the remainder of the request or, if the request has already been completed, ignore this response. The server MUST send a final response after the request has been completed. See section 8.2.3 for detailed discussion of the use and handling of this status code.

Wikipedia

This means that the server has received the request headers, and that the client should proceed to send the request body (in the case of a request for which a body needs to be sent; for example, a POST request). If the request body is large, sending it to a server when a request has already been rejected based upon inappropriate headers is inefficient. To have a server check if the request could be accepted based on the request's headers alone, a client must send Expect: 100-continue as a header in its initial request and check if a 100 Continue status code is received in response before continuing (or receive 417 Expectation Failed and not continue).

101 Switching Protocols

The server understands and is willing to comply with the client's request, via the Upgrade message header field (section 14.42), for a change in the application protocol being used on this connection. The server will switch protocols to those defined by the response's Upgrade header field immediately after the empty line which terminates the 101 response.

The protocol SHOULD be switched only when it is advantageous to do so. For example, switching to a newer version of HTTP is advantageous over older versions, and switching to a real-time, synchronous protocol might be advantageous when delivering resources that use such features.

Wikipedia

This means the requester has asked the server to switch protocols and the server is acknowledging that it will do so.

102 Processing (WebDAV)

The 102 (Processing) status code is an interim response used to inform the client that the server has accepted the complete request, but has not yet completed it. This status code SHOULD only be sent when the server has a reasonable expectation that the request will take significant time to complete. As guidance, if a method is taking longer than 20 seconds (a reasonable, but arbitrary value) to process the server SHOULD return a 102 (Processing) response. The server MUST send a final response after the request has been completed.

Methods can potentially take a long period of time to process, especially methods that support the Depth header. In such cases the client may time-out the connection while waiting for a response. To prevent this the server may return a 102 (Processing) status code to indicate to the client that the server is still processing the method.

Wikipedia

As a WebDAV request may contain many sub-requests involving file operations, it may take a long time to complete the request. This code indicates that the server has received and is processing the request, but no response is available yet. This prevents the client from timing out and assuming the request was lost.

2xx Success

***** 200 OK

The request has succeeded. The information returned with the response is dependent on the method used in the request, for example:

- GET an entity corresponding to the requested resource is sent in the response;
- HEAD the entity-header fields corresponding to the requested resource are sent in the response without any message-body;
- POST an entity describing or containing the result of the action;
- TRACE an entity containing the request message as received by the end server.

Wikipedia

Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain an entity corresponding to the requested resource. In a POST request the response will contain an entity describing or containing the result of the action.

***** General status code. Most common code used to indicate success.

***** 201 Created

The request has been fulfilled and resulted in a new resource being created. The newly created resource can be referenced by the URI(s) returned in the entity of the response, with the most specific URI for the resource given by a Location header field. The response SHOULD include an entity containing a list of resource characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. The origin server MUST create the resource before returning the 201 status code. If the action cannot be carried out immediately, the server SHOULD respond with 202 (Accepted) response instead.

A 201 response MAY contain an ETag response header field indicating the current value of the entity tag for the requested variant just created, see section 14.19.

Wikipedia

The request has been fulfilled and resulted in a new resource being created.

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

202 Accepted

The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place. There is no facility for re-sending a status code from an asynchronous operation such as this.

The 202 response is intentionally non-committal. Its purpose is to allow a server to accept a request for some other process (perhaps a batch-oriented process that is only run once per day) without requiring that the user agent's connection to the server persist until the process is completed. The entity returned with this response SHOULD include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the user can expect the request to be fulfilled.

Wikipedia

The request has been accepted for processing, but the processing has not been completed. The request might or might not eventually be acted upon, as it might be disallowed when processing actually takes place.

203 Non-Authoritative Information

The returned metainformation in the entity-header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy. The set presented MAY be a subset or superset of the original version. For example, including local annotation information about the resource might result in a superset of the metainformation known by the origin server. Use of this response code is not required and is only appropriate when the response would otherwise be 200 (OK).

Wikipedia

The server successfully processed the request, but is returning information that may be from another source.

Not present in HTTP/1.0: available since HTTP/1.1

***** 204 No Content

The server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation. The response MAY include new or updated metainformation in the form of entity-headers, which if present SHOULD be associated with the requested variant.

If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent. This response is primarily intended to allow input for actions to take place without causing a change to the user agent's active document view, although any new or updated metainformation SHOULD be applied to the document currently in the user agent's active view.

The 204 response MUST NOT include a message-body, and thus is always terminated by the first empty line after the header fields.

Wikipedia

The server successfully processed the request, but is not returning any content.

***** Status when wrapped responses (e.g. JSEND) are not used and nothing is in the body (e.g. DELETE).

205 Reset Content

The server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent. This response is primarily intended to allow input for actions to take place via user input, followed by a clearing of the form in which the input is given so that the user can easily initiate another input action. The response MUST NOT include an entity.

Wikipedia

The server successfully processed the request, but is not returning any content. Unlike a 204 response, this response requires that the requester reset the document view.

206 Partial Content

The server has fulfilled the partial GET request for the resource. The request MUST have included a Range header field (section 14.35) indicating the desired range, and MAY have included an If-Range header field (section 14.27) to make the request conditional.

The response MUST include the following header fields:

- Either a Content-Range header field (section 14.16) indicating the range included with this response, or a multipart/byteranges Content-Type including Content-Range fields for each part. If a Content-Length header field is present in the response, its value MUST match the actual number of OCTETs transmitted in the message-body.
- Date
- ETag and/or Content-Location, if the header would have been sent in a 200 response to the same request
- Expires, Cache-Control, and/or Vary, if the field-value might differ from that sent in any previous response for the same variant

If the 206 response is the result of an If-Range request that used a strong cache validator (see section 13.3.3), the response SHOULD NOT include other entity-headers. If the response is the result of an If-Range request that used a weak validator, the response MUST NOT include other entity-headers; this prevents inconsistencies between cached entity-bodies and updated headers. Otherwise, the response MUST include all of the entity-headers that would have been returned with a 200 (OK) response to the same request.

A cache MUST NOT combine a 206 response with other previously cached content if the ETag or Last-Modified headers do not match exactly, see 13.5.4.

A cache that does not support the Range and Content-Range headers MUST NOT cache 206 (Partial) responses.

Wikipedia

The server is delivering only part of the resource due to a range header sent by the client. The range header is used by tools like wget to enable resuming of interrupted downloads, or split a download into multiple simultaneous streams.

207 Multi-Status (WebDAV)

The 207 (Multi-Status) status code provides status for multiple independent operations (see section 11 for more information).

Wikipedia

The message body that follows is an XML message and can contain a number of separate response codes, depending on how many sub-requests were made.

208 Already Reported (WebDAV)

The 208 (Already Reported) status code can be used inside a DAV: propstat response element to avoid enumerating the internal members of multiple bindings to the same collection repeatedly. For each

binding to a collection inside the request's scope, only one will be reported with a 200 status, while subsequent DAV:response elements for all other bindings will use the 208 status, and no DAV:response elements for their descendants are included.

Wikipedia

The members of a DAV binding have already been enumerated in a previous reply to this request, and are not being included again.

226 IM Used

The server has fulfilled a GET request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance. The actual current instance might not be available except by combining this response with other previous or future responses, as appropriate for the specific instance-manipulation(s). If so, the headers of the resulting instance are the result of combining the headers from the status-226 response and the other instances, following the rules in section 13.5.3 of the HTTP/1.1 specification.

The request MUST have included an A-IM header field listing at least one instance-manipulation. The response MUST include an Etag header field giving the entity tag of the current instance.

A response received with a status code of 226 MAY be stored by a cache and used in reply to a subsequent request, subject to the HTTP expiration mechanism and any Cache-Control headers, and to the requirements in section 10.6.

A response received with a status code of 226 MAY be used by a cache, in conjunction with a cache entry for the base instance, to create a cache entry for the current instance.

Wikipedia

The server has fulfilled a GET request for the resource, and the response is a representation of the result of one or more instance-manipulations applied to the current instance.

3xx Redirection

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request. The action required MAY be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD. A client SHOULD detect infinite redirection loops, since such loops generate network traffic for each redirection.

Note: previous versions of this specification recommended a maximum of five redirections. Content developers should be aware that there might be clients that implement such a fixed limitation.

Wikipedia

The client must take additional action to complete the request.

This class of status code indicates that further action needs to be taken by the user agent in order to fulfil the request. The action required may be carried out by the user agent without interaction with the user if and only if the method used in the second request is GET or HEAD. A user agent should not automatically redirect a request more than five times, since such redirections usually indicate an infinite loop.

300 Multiple Choices

The requested resource corresponds to any one of a set of representations, each with its own specific location, and agent- driven negotiation information (section 12) is being provided so that the user (or user agent) can select a preferred representation and redirect its request to that location.

Unless it was a HEAD request, the response SHOULD include an entity containing a list of resource characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content- Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice MAY be performed automatically. However, this specification does not define any standard for such automatic selection.

If the server has a preferred choice of representation, it SHOULD include the specific URI for that representation in the Location field; user agents MAY use the Location field value for automatic redirection. This response is cacheable unless indicated otherwise.

Wikipedia

Indicates multiple options for the resource that the client may follow. It, for instance, could be used to present different format options for video, list files with different extensions, or word sense disambiguation.

301 Moved Permanently

The requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs. Clients with link editing capabilities ought to automatically re-link references to the Request-URI to one or more of the new references returned by the server, where possible. This response is cacheable unless indicated otherwise.

The new permanent URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

If the 301 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Note: When automatically redirecting a POST request after receiving a 301 status code, some existing HTTP/1.0 user agents will erroneously change it into a GET request.

Wikipedia

This and all future requests should be directed to the given URI.

302 Found

The requested resource resides temporarily under a different URI. Since the redirection might be altered on occasion, the client SHOULD continue to use the Request-URI for future requests. This response is only cacheable if indicated by a Cache-Control or Expires header field.

The temporary URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

If the 302 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Note: RFC 1945 and RFC 2068 specify that the client is not allowed to change the method on the redirected request. However, most existing user agent implementations treat 302 as if it were a 303 response, performing a GET on the Location field-value regardless of the original request method. The status codes 303 and 307 have been added for servers that wish to make unambiguously clear which kind of reaction is expected of the client.

Wikipedia

This is an example of industry practice contradicting the standard.[2] The HTTP/1.0 specification (RFC 1945) required the client to perform a temporary redirect (the original describing phrase was "Moved Temporarily"), but popular browsers implemented 302 with the functionality of a 303 See Other. Therefore, HTTP/1.1 added status codes 303 and 307 to distinguish between the two behaviours. However, some Web applications and frameworks use the 302 status code as if it were the 303.

303 See Other

The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new URI is not a substitute reference for the originally requested resource. The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

The different URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

Note: Many pre-HTTP/1.1 user agents do not understand the 303 status. When interoperability with such clients is a concern, the 302 status code may be used instead, since most user agents react to a 302 response as described here for 303.

Wikipedia

The response to the request can be found under another URI using a GET method. When received in response to a POST (or PUT/DELETE), it should be assumed that the server has received the data and the redirect should be issued with a separate GET message.

Since HTTP/1.1

***** 304 Not Modified

If the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code. The 304 response MUST NOT contain a message-body, and thus is always terminated by the first empty line after the header fields.

The response MUST include the following header fields:

- Date, unless its omission is required by section 14.18.1

If a clockless origin server obeys these rules, and proxies and clients add their own Date to any response received without one (as already specified by [RFC 2068], section 14.19), caches will operate correctly.

- ETag and/or Content-Location, if the header would have been sent in a 200 response to the same request
- Expires, Cache-Control, and/or Vary, if the field-value might differ from that sent in any previous response for the same variant

If the conditional GET used a strong cache validator (see section 13.3.3), the response SHOULD NOT include other entity-headers. Otherwise (i.e., the conditional GET used a weak validator), the response MUST NOT include other entity-headers; this prevents inconsistencies between cached entity-bodies and updated headers.

If a 304 response indicates an entity not currently cached, then the cache MUST disregard the response and repeat the request without the conditional.

If a cache uses a received 304 response to update a cache entry, the cache MUST update the entry to reflect any new field values given in the response.

Wikipedia

Indicates the resource has not been modified since last requested. Typically, the HTTP client provides a header like the If-Modified-Since header to provide a time against which to compare. Using this saves bandwidth and reprocessing on both the server and client, as only the header data must be sent and received in comparison to the entirety of the page being re-processed by the server, then sent again using more bandwidth of the server and client.

***** 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 on the response.

305 Use Proxy

The requested resource MUST be accessed through the proxy given by the Location field. The Location field gives the URI of the proxy. The recipient is expected to repeat this single request via the proxy. 305 responses MUST only be generated by origin servers.

Note: RFC 2068 was not clear that 305 was intended to redirect a single request, and to be generated by origin servers only. Not observing these limitations has significant security consequences.

Wikipedia

Many HTTP clients (such as Mozilla and Internet Explorer) do not correctly handle responses with this status code, primarily for security reasons.

306 (Unused)

The 306 status code was used in a previous version of the specification, is no longer used, and the code is reserved.

Wikipedia

No longer used. Originally meant "Subsequent requests should use the specified proxy."

307 Temporary Redirect

The requested resource resides temporarily under a different URI. Since the redirection MAY be altered on occasion, the client SHOULD continue to use the Request-URI for future requests. This response is only cacheable if indicated by a Cache-Control or Expires header field.

The temporary URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s) , since many pre-HTTP/1.1 user agents do not understand the 307 status. Therefore, the note SHOULD contain the information necessary for a user to repeat the original request on the new URI.

If the 307 status code is received in response to a request other than GET or HEAD, the user agent MUST NOT automatically redirect the request unless it can be confirmed by the user, since this might change the conditions under which the request was issued.

Wikipedia

In this case, the request should be repeated with another URI; however, future requests can still use the original URI. In contrast to 302, the request method should not be changed when reissuing the original request. For instance, a POST request must be repeated using another POST request.

308 Permanent Redirect (experiemental)

Wikipedia

The request, and all future requests should be repeated using another URI. 307 and 308 (as proposed) parallel the behaviours of 302 and 301, but do not require the HTTP method to change. So, for example, submitting a form to a permanently redirected resource may continue smoothly.

4xx Client Error

The 4xx class of status code is intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents SHOULD display any included entity to the user.

If the client is sending data, a server implementation using TCP SHOULD be careful to ensure that the client acknowledges receipt of the packet(s) containing the response, before the server closes the input connection. If the client continues sending data to the server after the close, the server's TCP stack will send a reset packet to the client, which may erase the client's unacknowledged input buffers before they can be read and interpreted by the HTTP application.

Wikipedia

The 4xx class of status code is intended for cases in which the client seems to have erred. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. These status codes are applicable to any request method. User agents should display any included entity to the user.

* 400 Bad Request

The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.

Wikipedia

The request cannot be fulfilled due to bad syntax.

***** General error when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

***** 401 Unauthorized
The request requires user authentication. The response MUST include a WWW-Authenticate header field (section 14.47) containing a challenge applicable to the requested resource. The client MAY repeat the request with a suitable Authorization header field (section 14.8). If the request already included Authorization credentials, then the 401 response indicates that authorization has been refused for those credentials. If the 401 response contains the same challenge as the prior response, and the user agent has already attempted authentication at least once, then the user SHOULD be presented the entity that was given in the response, since that entity might include relevant diagnostic information. HTTP access authentication is explained in "HTTP Authentication: Basic and Digest Access Authentication".

Wikipedia
Similar to 403 Forbidden, but specifically for use when authentication is possible but has failed or not yet been provided. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource. See Basic access authentication and Digest access authentication.

***** Error code response for missing or invalid authentication token.

402 Payment Required
This code is reserved for future use.

Wikipedia
Reserved for future use. The original intention was that this code might be used as part of some form of digital cash or micropayment scheme, but that has not happened, and this code is not usually used. As an example of its use, however, Apple's MobileMe service generates a 402 error ("httpStatusCode:402" in the Mac OS X Console log) if the MobileMe account is delinquent.

403 Forbidden
The server understood the request, but is refusing to fulfill it. Authorization will not help and the request SHOULD NOT be repeated. If the request method was not HEAD and the server wishes to make public why the request has not been fulfilled, it SHOULD describe the reason for the refusal in the entity. If the server does not wish to make this information available to the client, the status code 404 (Not Found) can be used instead.

Wikipedia
The request was a legal request, but the server is refusing to respond to it. Unlike a 401 Unauthorized response, authenticating will make no difference.

***** 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
The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

Wikipedia

The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.

***** 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.

405 Method Not Allowed

The method specified in the Request-Line is not allowed for the resource identified by the Request-URI. The response MUST include an Allow header containing a list of valid methods for the requested resource.

Wikipedia

A request was made of a resource using a request method not supported by that resource; for example, using GET on a form which requires data to be presented via POST, or using PUT on a read-only resource.

406 Not Acceptable

The resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

Unless it was a HEAD request, the response SHOULD include an entity containing a list of available entity characteristics and location(s) from which the user or user agent can choose the one most appropriate. The entity format is specified by the media type given in the Content-Type header field. Depending upon the format and the capabilities of the user agent, selection of the most appropriate choice MAY be performed automatically. However, this specification does not define any standard for such automatic selection.

Note: HTTP/1.1 servers are allowed to return responses which are not acceptable according to the accept headers sent in the request. In some cases, this may even be preferable to sending a 406 response. User agents are encouraged to inspect the headers of an incoming response to determine if it is acceptable.

If the response could be unacceptable, a user agent SHOULD temporarily stop receipt of more data and query the user for a decision on further actions.

Wikipedia

The requested resource is only capable of generating content not acceptable according to the Accept headers sent in the request.

407 Proxy Authentication Required

This code is similar to 401 (Unauthorized), but indicates that the client must first authenticate itself with the proxy. The proxy MUST return a Proxy-Authenticate header field (section 14.33) containing a challenge applicable to the proxy for the requested resource. The client MAY repeat the request with a suitable Proxy-Authorization header field (section 14.34). HTTP access authentication is explained in "HTTP Authentication: Basic and Digest Access Authentication".

Wikipedia

The client must first authenticate itself with the proxy.

408 Request Timeout

The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time.

Wikipedia

The server timed out waiting for the request. According to W3 HTTP specifications: "The client did not produce a request within the time that the server was prepared to wait. The client MAY repeat the request without modifications at any later time."

***** 409 Conflict

The request could not be completed due to a conflict with the current state of the resource. This code is only allowed in situations where it is expected that the user might be able to resolve the conflict and resubmit the request. The response body SHOULD include enough information for the user to recognize the source of the conflict. Ideally, the response entity would include enough information for the user or user agent to fix the problem; however, that might not be possible and is not required.

Conflicts are most likely to occur in response to a PUT request. For example, if versioning were being used and the entity being PUT included changes to a resource which conflict with those made by an earlier (third-party) request, the server might use the 409 response to indicate that it can't complete the request. In this case, the response entity would likely contain a list of the differences between the two versions in a format defined by the response Content-Type.

Wikipedia

Indicates that the request could not be processed because of conflict in the request, such as an edit conflict.

***** Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries and deleting root objects when cascade-delete is not supported are a couple of examples.

410 Gone

The requested resource is no longer available at the server and no forwarding address is known. This condition is expected to be considered permanent. Clients with link editing capabilities SHOULD delete references to the Request-URI after user approval. If the server does not know, or has no facility to determine, whether or not the condition is permanent, the status code 404 (Not Found) SHOULD be used instead. This response is cacheable unless indicated otherwise.

The 410 response is primarily intended to assist the task of web maintenance by notifying the recipient that the resource is intentionally unavailable and that the server owners desire that remote links to that resource be removed. Such an event is common for limited-time, promotional services and for resources belonging to individuals no longer working at the server's site. It is not necessary to mark all permanently unavailable resources as "gone" or to keep the mark for any length of time -- that is left to the discretion of the server owner.

Wikipedia

Indicates that the resource requested is no longer available and will not be available again. This should be used when a resource has been intentionally removed and the resource should be purged. Upon receiving a 410 status code, the client should not request the resource again in the future. Clients such as search engines should remove the resource from their indices. Most use cases do not require clients and search engines to purge the resource, and a "404 Not Found" may be used instead.

411 Length Required

The server refuses to accept the request without a defined Content- Length. The client MAY repeat the request if it adds a valid Content-Length header field containing the length of the message-body in the request message.

Wikipedia

The request did not specify the length of its content, which is required by the requested resource.

412 Precondition Failed

The precondition given in one or more of the request-header fields evaluated to false when it was tested on the server. This response code allows the client to place preconditions on the current resource metainformation (header field data) and thus prevent the requested method from being applied to a resource other than the one intended.

Wikipedia

The server does not meet one of the preconditions that the requester put on the request.

413 Request Entity Too Large

The server is refusing to process a request because the request entity is larger than the server is willing or able to process. The server MAY close the connection to prevent the client from continuing the request.

If the condition is temporary, the server SHOULD include a Retry- After header field to indicate that it is temporary and after what time the client MAY try again.

Wikipedia

The request is larger than the server is willing or able to process.

414 Request-URI Too Long

The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret. This rare condition is only likely to occur when a client has improperly converted a POST request to a GET request with long query information, when the client has descended into a URI "black hole" of redirection (e.g., a redirected URI prefix that points to a suffix of itself), or when the server is under attack by a client attempting to exploit security holes present in some servers using fixed-length buffers for reading or manipulating the Request-URI.

Wikipedia

The URI provided was too long for the server to process.

415 Unsupported Media Type

The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

Wikipedia

The request entity has a media type which the server or resource does not support. For example, the client uploads an image as image/svg+xml, but the server requires that images use a different format.

416 Requested Range Not Satisfiable

A server SHOULD return a response with this status code if a request included a Range request-header field (section 14.35), and none of the range-specifier values in this field overlap the current extent of the selected resource, and the request did not include an If-Range request-header field. (For byte-ranges, this means that the first- byte-pos of all of the byte-range-spec values were greater than the current length of the selected resource.)

When this status code is returned for a byte-range request, the response SHOULD include a Content-Range entity-header field specifying the current length of the selected resource (see section 14.16). This response MUST NOT use the multipart/byteranges content- type.

Wikipedia

The client has asked for a portion of the file, but the server cannot supply that portion. For example, if the client asked for a part of the file that lies beyond the end of the file.

417 Expectation Failed

The expectation given in an Expect request-header field (see section 14.20) could not be met by this server, or, if the server is a proxy, the server has unambiguous evidence that the request could not be met by the next-hop server.

Wikipedia

The server cannot meet the requirements of the Expect request-header field.

418 I'm a teapot (RFC 2324)

Wikipedia

This code was defined in 1998 as one of the traditional IETF April Fools' jokes, in RFC 2324, Hyper Text Coffee Pot Control Protocol, and is not expected to be implemented by actual HTTP servers. However, known implementations do exist. An Nginx HTTP server uses this code to simulate goto-like behaviour in its configuration.

420 Enhance Your Calm (Twitter)

Wikipedia

Returned by the Twitter Search and Trends API when the client is being rate limited. Likely a reference to this number's association with marijuana. Other services may wish to implement the 429 Too Many Requests response code instead.

422 Unprocessable Entity (WebDAV)

The 422 (Unprocessable Entity) status code means the server understands the content type of the request entity (hence a 415(Unsupported Media Type) status code is inappropriate), and the syntax of the request entity is correct (thus a 400 (Bad Request) status code is inappropriate) but was unable to process the contained instructions. For example, this error condition may occur if an XML request body contains well-formed (i.e., syntactically correct), but semantically erroneous, XML instructions.

Wikipedia

The request was well-formed but was unable to be followed due to semantic errors.

423 Locked (WebDAV)

The 423 (Locked) status code means the source or destination resource of a method is locked. This response SHOULD contain an appropriate precondition or postcondition code, such as 'lock-token-submitted' or 'no-conflicting-lock'.

Wikipedia

The resource that is being accessed is locked.

424 Failed Dependency (WebDAV)

The 424 (Failed Dependency) status code means that the method could not be performed on the resource because the requested action depended on another action and that action failed. For example, if a command in a PROPPATCH method fails, then, at minimum, the rest of the commands will also fail with 424 (Failed Dependency).

Wikipedia

The request failed due to failure of a previous request (e.g. a PROPPATCH).

425 Reserved for WebDAV

Slein, J., Whitehead, E.J., et al., "WebDAV Advanced Collections Protocol", Work In Progress.

Wikipedia

Defined in drafts of "WebDAV Advanced Collections Protocol", but not present in "Web Distributed Authoring and Versioning (WebDAV) Ordered Collections Protocol".

426 Upgrade Required

Reliable, interoperable negotiation of Upgrade features requires an unambiguous failure signal. The 426 Upgrade Required status code allows a server to definitively state the precise protocol extensions a given resource must be served with.

Wikipedia

The client should switch to a different protocol such as TLS/1.0.

428 Precondition Required

The 428 status code indicates that the origin server requires the request to be conditional.

Its typical use is to avoid the "lost update" problem, where a client GETs a resource's state, modifies it, and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a conflict. By requiring requests to be conditional, the server can assure that clients are working with the correct copies.

Responses using this status code SHOULD explain how to resubmit the request successfully.

The 428 status code is optional; clients cannot rely upon its use to prevent "lost update" conflicts.

Wikipedia

The origin server requires the request to be conditional. Intended to prevent "the "lost update" problem, where a client GETs a resource's state, modifies it, and PUTs it back to the server, when meanwhile a third party has modified the state on the server, leading to a conflict.

429 Too Many Requests

The 429 status code indicates that the user has sent too many requests in a given amount of time ("rate limiting").

The response representations SHOULD include details explaining the condition, and MAY include a Retry-After header indicating how long to wait before making a new request.

When a server is under attack or just receiving a very large number of requests from a single party, responding to each with a 429 status code will consume resources.

Therefore, servers are not required to use the 429 status code; when limiting resource usage, it may be more appropriate to just drop connections, or take other steps.

Wikipedia

The user has sent too many requests in a given amount of time. Intended for use with rate limiting schemes.

431 Request Header Fields Too Large

The 431 status code indicates that the server is unwilling to process the request because its header fields are too large. The request MAY be resubmitted after reducing the size of the request header fields.

It can be used both when the set of request header fields in total are too large, and when a single header field is at fault. In the latter case, the response representation SHOULD specify which header field was too large.

Servers are not required to use the 431 status code; when under attack, it may be more appropriate to just drop connections, or take other steps.

Wikipedia

The server is unwilling to process the request because either an individual header field, or all the header fields collectively, are too large.

444 No Response (Nginx)

Wikipedia

An Nginx HTTP server extension. The server returns no information to the client and closes the connection (useful as a deterrent for malware).

449 Retry With (Microsoft)

Wikipedia

A Microsoft extension. The request should be retried after performing the appropriate action.

450 Blocked by Windows Parental Controls (Microsoft)

Wikipedia

A Microsoft extension. This error is given when Windows Parental Controls are turned on and are blocking access to the given webpage.

451 Unavailable For Legal Reasons

Wikipedia

Intended to be used when resource access is denied for legal reasons, e.g. censorship or government-mandated blocked access. A reference to the 1953 dystopian novel Fahrenheit 451, where books are outlawed, and the autoignition temperature of paper, 451°F.

499 Client Closed Request (Nginx)

Wikipedia

An Nginx HTTP server extension. This code is introduced to log the case when the connection is closed by client while HTTP server is processing its request, making server unable to send the HTTP header back.

5xx Server Error

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request. Except when responding to a HEAD request, the server SHOULD include an entity containing an explanation of the error situation, and whether it is a temporary or permanent condition. User agents SHOULD display any included entity to the user. These response codes are applicable to any request method.

Wikipedia

The server failed to fulfill an apparently valid request.

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has encountered an error or is otherwise incapable of performing the request. Except when responding to a HEAD request, the server should include an entity containing an explanation of the error situation, and indicate whether it is a temporary or permanent condition. Likewise, user agents should display any included entity to the user. These response codes are applicable to any request method.

* 500 Internal Server Error

The server encountered an unexpected condition which prevented it from fulfilling the request.

Wikipedia

A generic error message, given when no more specific message is suitable.

* The general catch-all error when the server-side throws an exception.

501 Not Implemented

The server does not support the functionality required to fulfill the request. This is the appropriate response when the server does not recognize the request method and is not capable of supporting it for any resource.

Wikipedia

The server either does not recognise the request method, or it lacks the ability to fulfill the request.

502 Bad Gateway

The server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

Wikipedia

The server was acting as a gateway or proxy and received an invalid response from the upstream server.

503 Service Unavailable

The server is currently unable to handle the request due to a temporary overloading or maintenance of the server. The implication is that this is a temporary condition which will be alleviated after some delay. If known, the length of the delay MAY be indicated in a Retry-After header. If no Retry-After is given, the client SHOULD handle the response as it would for a 500 response.

Note: The existence of the 503 status code does not imply that a server must use it when becoming overloaded. Some servers may wish to simply refuse the connection.

Wikipedia

The server is currently unavailable (because it is overloaded or down for maintenance). Generally, this is a temporary state.

504 Gateway Timeout

The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request.

Note: Note to implementors: some deployed proxies are known to return 400 or 500 when DNS lookups time out.

Wikipedia

The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.

505 HTTP Version Not Supported

The server does not support, or refuses to support, the HTTP protocol version that was used in the request message. The server is indicating that it is unable or unwilling to complete the request using the same major version as the client, as described in section 3.1, other than with this error message. The response SHOULD contain an entity describing why that version is not supported and what other protocols are supported by that server.

Wikipedia

The server does not support the HTTP protocol version used in the request.

506 Variant Also Negotiates (Experimental)

The 506 status code indicates that the server has an internal configuration error: the chosen variant resource is configured to engage in transparent content negotiation itself, and is therefore not a proper end point in the negotiation process.

Wikipedia

Transparent content negotiation for the request results in a circular reference.

507 Insufficient Storage (WebDAV)

The 507 (Insufficient Storage) status code means the method could not be performed on the resource because the server is unable to store the representation needed to successfully complete the request. This condition is considered to be temporary. If the request that received this status code was the result of a user action, the request MUST NOT be repeated until it is requested by a separate user action.

Wikipedia
The server is unable to store the representation needed to complete the request.

508 Loop Detected (WebDAV)
The 508 (Loop Detected) status code indicates that the server terminated an operation because it encountered an infinite loop while processing a request with "Depth: infinity". This status indicates that the entire operation failed.

Wikipedia
The server detected an infinite loop while processing the request (sent in lieu of 208).

509 Bandwidth Limit Exceeded (Apache)
Wikipedia
This status code, while used by many servers, is not specified in any RFCs.

510 Not Extended
The policy for accessing the resource has not been met in the request. The server should send back all the information necessary for the client to issue an extended request. It is outside the scope of this specification to specify how the extensions inform the client.

If the 510 response contains information about extensions that were not present in the initial request then the client MAY repeat the request if it has reason to believe it can fulfill the extension policy by modifying the request according to the information provided in the 510 response. Otherwise the client MAY present any entity included in the 510 response to the user, since that entity may include relevant diagnostic information.

Wikipedia
Further extensions to the request are required for the server to fulfill it.

511 Network Authentication Required
The 511 status code indicates that the client needs to authenticate to gain network access.

The response representation SHOULD contain a link to a resource that allows the user to submit credentials (e.g. with a HTML form).

Note that the 511 response SHOULD NOT contain a challenge or the login interface itself, because browsers would show the login interface as being associated with the originally requested URL, which may cause confusion.

The 511 status SHOULD NOT be generated by origin servers; it is intended for use by intercepting proxies that are interposed as a means of controlling access to the network.

Responses with the 511 status code MUST NOT be stored by a cache.

The 511 status code is designed to mitigate problems caused by "captive portals" to software (especially non-browser agents) that is expecting a response from the server that a request was made to, not the intervening network infrastructure. It is not intended to encouraged deployment of captive portals, only to limit the damage caused by them.

A network operator wishing to require some authentication, acceptance of terms or other user interaction before granting access usually does so by identifing clients who have not done so ("unknown clients") using their MAC addresses.

Unknown clients then have all traffic blocked, except for that on TCP port 80, which is sent to a HTTP server (the "login server") dedicated to "logging in" unknown clients, and of course traffic to the login server itself.

In common use, a response carrying the 511 status code will not come from the origin server indicated in the request's URL. This presents many security issues; e.g., an attacking intermediary may be inserting cookies into the original domain's name space, may be observing cookies or HTTP authentication credentials sent from the user agent, and so on.

However, these risks are not unique to the 511 status code; in other words, a captive portal that is not using this status code introduces the same issues.

Also, note that captive portals using this status code on an SSL or TLS connection (commonly, port 443) will generate a certificate error on the client.

Wikipedia

The client needs to authenticate to gain network access. Intended for use by intercepting proxies used to control access to the network (e.g., "captive portals" used to require agreement to Terms of Service before granting full Internet access via a Wi-Fi hotspot).

598 Network read timeout error

Wikipedia

This status code is not specified in any RFCs, but is used by some HTTP proxies to signal a network read timeout behind the proxy to a client in front of the proxy.

599 Network connect timeout error

Wikipedia

This status code is not specified in any RFCs, but is used by some HTTP proxies to signal a network connect timeout behind the proxy to a client in front of the proxy.

*** "Top 10"** HTTP Status Code. More REST service-specific information is contained in the entry.

REST API Resources

Translations

Russian

http://www.restapitutorial.ru/

REST API Cheat Sheets

- API Design Cheat Sheet - This GitHub repository outlines important tips to consider when designing APIs that developers love.
- Platform-Building Cheat Sheet - Everyone wants to build a platform. This GitHub repository is a public receptical of ground rules when building a platform.

REST API Best Practices

Get the *RESTful Best Practices* guide (choose your format). This guide reduces the world of RESTful services into easy-to-follow principles. It also provides several cookbook type recipes in critical areas to increase service usability, reduce confusion during implemenation, as well as improve consistency.

- PDF (~306KB)
- ePub (~46KB). Works on iPad, iPhone, B&N Nook and most other readers.
- Mobi (~86KB). Works on Kindle, Kindle Reader Apps
- Source Document in Libre/Open Office format (~48KB)

Building REST APIs in Java

RestExpress (GitHub). A microservices framework for Java, RestExpress composes best-of-breed tools to form a lightweight, minimalist Java framework for quickly creating RESTful APIs.

Web Resources

- REST API Tutorial YouTube Channel
- Todd Fredrich's Blog