12/4/2020, 1/25/21, 4/13/21

**RE**presentational **“State”** **T**ransfer: **REST**

1. **Content Negotiation using HTTP headers**

most REST API implementations rely on **agent-driven content negotiations**. Agent driven content negotiation rely on usage of **HTTP request headers** or **resource URI patterns**.

At server side, an incoming request may have an entity attached to it. To determine its type, server uses the HTTP request header Content-Type. Some common examples of content types are

“text/plain”, “text/html”,

“application/json”, “application/xml”,

“image/gif”, and “image/jpeg”.

**Content-Type**: application/json

Similarly, to determine what type of representation is desired at client side, HTTP header ACCEPT is used. It will have one of the values as mentioned for Content-Type above.

**Accept**: application/json

**Client side request**

Generally, if no Accept header is present in the request, the server can send pre-configured default representation type.

*Implementing Accept header based content negotiation is most used and recommended way.*

**Content negotiation using URL patterns**

Another way to pass content type information to the server, the client may use the **specific extension** in **resource URIs**. For example, a client can ask for details using:

http://rest.api.com/v1/employees/20423.xml

http://rest.api.com/v1/employees/20423.json

In above case, first request URI will return a XML response whether second request URI will return a JSON response.

## **Defining preferences**

It is possible to have multiple values in Accept header. The client may want to give multiple values in the accept header when the client is not sure about if its desired representation is present or supported by the server at that time. [[RFC 2296](https://tools.ietf.org/html/rfc2296)]

For example,

Accept: application/json,application/xml;q=0.9,\*/\*;q=0.8

Above Accept header allows you to ask the server a JSON format. If it can’t, perhaps it could return XML format (the second level). If it’s still not possible, let it return what it can.

The preference order is defined through the q parameter with values from 0 to 1. When nothing is specified, the implicit value is 1.

1. **Idempotent REST APIs**

In the context of REST APIs, **when making multiple identical requests** has t**he same effect as making a single request** – then that REST API is called **idempotent**.

When you design REST APIs, you must realize that API consumers can make mistakes. Users can write client code in such a way that there can be duplicate requests coming to the API.

These duplicate requests may be unintentional as well as intentional some time (e.g. due to timeout or network issues). You have to design fault-tolerant APIs in such a way that **duplicate requests** do not leave the system unstable.

**An idempotent HTTP method** *is an HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same.*

*Idempotence essentially means that the result of a successfully performed request is independent of the number of times it is executed. For example, in arithmetic, adding zero to a number is an idempotent operation.*

## **Idempotency with HTTP Methods**

If you follow REST principles in designing API, you will have automatically **idempotent REST APIs** for GET, PUT, DELETE, HEAD, OPTIONS and TRACE HTTP methods. Only POST APIs will not be idempotent.

1. **POST, Patch is NOT idempotent.**
2. **GET, PUT, DELETE, HEAD, OPTIONS and TRACE are idempotent.**

Let’s analyze how the above HTTP methods end up being idempotent – and why POST is not.

#### HTTP POST

Generally – not necessarily – POST APIs are used to create a new resource on server. So when you invoke the same POST request N times, you will have N new resources on the server. So, **POST is not idempotent**.

#### HTTP GET, HEAD, OPTIONS and TRACE

#### GET, HEAD, OPTIONS and TRACE methods NEVER change the resource state on server. They are purely for retrieving the resource representation or meta data at that point of time. So invoking multiple requests will not have any write operation on server, so **GET, HEAD, OPTIONS and TRACE are idempotent**.

**HTTP Patch**

It’s like POST not idempotent since it’s usually used to **partially update a resource** (for example just sending and setting the first name of a user profile, ignoring the state of the remaining data).

#### HTTP PUT

Generally – not necessarily – PUT APIs are used to update the resource state. If you invoke a PUT API N times, the very first request will update the resource; then rest N-1 requests will just overwrite the same resource state again and again – effectively not changing anything. Hence, **PUT is idempotent**.

#### HTTP DELETE

When you invoke N similar DELETE requests, first request will delete the resource and response will be 200 (OK) or 204 (No Content). Other N-1 requests will return 404 (Not Found). Clearly, the response is different from first request, but there is no change of state for any resource on server side because original resource is already deleted. So, **DELETE is idempotent**.

Please keep in mind if some systems may have DELETE APIs like this:

[Rfc 2616](https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html)

1. **Statelessness**

Refer to ***Application state***

As per the REST (REpresentational **“State”** Transfer) architecture, the **server does not store any state about the client session on the server-side**. This restriction is called **Statelessness**. Each request from the client to server must **contain all of the information** necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client. **client is responsible for storing and handling all application state-related information on client** side.

It also means that the client is responsible for sending any state information to the server whenever it is needed. There should not be any *session affinity* or *sticky sessions* on server.

*Statelessness means that every HTTP request happens in complete isolation. When the client makes an HTTP request, it includes all information necessary for the server to fulfill that request. The server never relies on information from previous requests. If that information was important, the client would have sent it again in this request.*

To enable clients to access these stateless APIs, its necessary that servers also should include every piece of information that the client may need to create the state on its side.

For becoming stateless, do not store even [authentication/authorization](https://restfulapi.net/security-essentials/) details of client. **Provide credentials with the request**. Each request MUST stand alone and should not be affected by the previous conversation happened from the same client in past.

## Application State vs Resource State

***Resource state*** is a state that needs to be persistent and survivable even after client disconnect/restart/session end/whatever. ***Application state*** should live on the **client and should be supplied with each client request for stateless REST**(if we are talking about REST architecture and planning to scale our application well).

How to distinguish application state and resource state?

If your are working on multistep purchasing process, probably, it's better to save **state of this process in your Resource state(database),** because you want this state to be saved permanently. Otherwise, your clients need to refill a lot of information after disconnect/restart/whatever. Of course you could do it in cookies for example(and it would be Application state), and this state can live after browser restart. But it has the two downsides: 1)This state unavailable on other user's devices, 2)If you are creating genuine REST service, cookies would complicate client's life, because not all clients operate cookies well(except browsers).

**REST statelessness means being free on application state.**

## **Advantages of Statelessness**

There are some very noticeable advantages for having **REST APIs stateless**.

1. Statelessness helps in **scaling the APIs** to millions of concurrent users by deploying it to multiple servers**. Any server** can handle any request because there is **no session related dependency**.
2. Being stateless makes REST APIs **less complex** – by removing all server-side state **synchronization logic**.
3. A stateless API is also **easy to**[**cache**](https://restfulapi.net/caching/) as well. Specific software can decide whether or not to cache the result of an HTTP request just by looking at that one request. There’s no nagging uncertainty that state from a previous request might affect the cacheability of this one. **It improves the performance of applications.**
4. The server never loses track of “where” each client is in the application because the client sends all necessary information with each request.
5. **Resources Representation Compression**

REST APIs can return the resource representations in several formats such as **XML**, **JSON**, **HTML**, or **even plain text**. All such forms can be compressed to a lesser number of bytes to save bandwidth over the network. **Different protocols use different techniques to enable compression** and notify the clients about the compression scheme – so that the client can decompress it before consuming the representations.

*Compression, like encryption, is something that happens to the resource representation in transit and must be undone before the client can use the representation.*

HTTP is most widely used protocol for REST – so I am taking example of **HTTP specific response compression**.

## **Compression Related Request/Response Headers**

### Accept-Encoding (request)

While requesting resource representations – along with an HTTP request, the **client** sends an **Accept-Encoding header (request)** that says what kind of compression algorithms the client understands.

The two standard values for Accept-Encoding are **compress** and **gzip**.

A sample request with accept-encoding header looks like this :

**GET /employees HTTP/1.1**

**Host: www.domain.com**

**Accept: text/html**

**Accept-Encoding: gzip, compress**

Other possible usage of accept-encoding may be:

Accept-Encoding: compress, gzip

Accept-Encoding:

Accept-Encoding: \*

Accept-Encoding: compress;q=0.5, gzip;q=1.0

Accept-Encoding: gzip;q=1.0, identity; q=0.5, \*;q=0

If an Accept-Encoding field is present in a request, and if the server cannot send a response which is acceptable according to the Accept-Encoding header, then the server SHOULD send an error response with the **406 (Not Acceptable)** status code.

**406 (Not Acceptable)**

### Content-Encoding (request, response)

If the server understands one of the compression algorithms from Accept-Encoding, it can use that algorithm to compress the representation before serving it. When successfully compressed, server lets know the client of encoding scheme by another HTTP header i.e. Content-Encoding.

200 OK

Content-Type: text/html

Content-Encoding: gzip

If the **content-type** of an entity in a **request** message is not acceptable to the origin server, the server SHOULD respond with a status code of **415 (Unsupported Media Type).** If multiple content encodings have been applied to an entity, all the encodings MUST be listed in the order in which they were used.

**415 (Unsupported Media Type)**

Please note that the original media-type for request and response are not impacted whether compression is requested or not.

Compression can save a lot of bandwidth, with minimal cost without additional complexity. Also, you may know that most web browsers automatically request compressed representations from website host servers – using the above headers.

1. **HTTP Status 204 (No Content)**

**https://tools.ietf.org/html/rfc7231#section-6.3.5**

HTTP Status 204 (No Content) indicates that the server has successfully fulfilled the request and that there is **no content to send in the response** payload body. The server might want to **return updated meta-information in the form of entity-headers**, which, if present, SHOULD be applied to the current document’s active view if any.

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

By default, 204 (No Content) response is **cacheable**. If caching needs to be overridden then response must include cache respective [cache headers](https://restfulapi.net/caching/).

For example, you may want to return status 204 (No Content) in **UPDATE operations** where **request payload is large enough not to transport back and forth**. The user agent will send the payload to the server to update the resource. If the operation is successful, the server will respond with 204 to indicate the success so that client application can update its UI to inform the user about the operation’s success.

It is also frequently used with interfaces that expect automated data transfers to be prevalent, such as within distributed version control systems.

## **Resolving lost update problem**

With **status 204**, server may also include **HTTP header ETag** to let the client validate client side resource representation before making further update on server – to avoid **lost update problem**.

**Lost update problem** happens when multiple people edit a resource without knowledge of each other’s changes. In this scenario, the last person to update a resource “wins,” and previous updates are lost. ETags can be used in combination with the If-Match header to let the server decide if a resource should be updated. If ETag does not match then server informs the client via a **412 (Precondition Failed)** response.

1. **HTTP Status** **301 (Moved Permanently)**

**https://tools.ietf.org/html/rfc7231#section-6.4.2**

HTTP Status 301 is one of the **redirection** related statuses, which indicates that the resource requested has been permanently moved to the URL given by the **Location header**. And all the future requests should use the new URI.

Redirection is the **process of forwarding the request from one URL to a different URL**. The specification for Status 301 requires the request method (and the request body) not to be altered when the redirection is performed.

It is recommended to use the **HTTP 301 only for GET or HEAD methods**.

## Why a URL needs to move permanently?

Generally, changing the URLs of resources is not advisable. Still, we can come across unavoidable situations where we must be making the changes to the URL of a resource.

A few such examples can be:

* **Moving the resource from HTTP to HTTPs protocol**
* Resource has been discountinued and alternate resource is available in new URL

## **Location Header**

The server SHOULD generate a Location header field in the response containing the new location of the resource.

#### **Client request**

|  |
| --- |
| **GET /index.php HTTP/1.1**  **Host: www.example.com** |

#### **Server response**

|  |
| --- |
| **HTTP/1.1 301 Moved Permanently**  **Location: https://example.com/index.asp** |

## **Cachable**

A 301 response is **cacheable by default**; i.e., unless otherwise indicated by the method definition or explicit cache controls.

Refer these [cache header](https://restfulapi.net/caching/) for more information.

## Response Handling

* If a client has link-editing capabilities, it should update all references to the requested URL with the new URL.
* Search engines (Google and Bing) replace the old URL in the search results, and the old URL will eventually disappear. Link juice will pass from the old URL to the new URL.
* The browsers will automatically detect the 301 response code after that it will read the new location URL and redirect the request to that new location.
* **If client got 301**, the client **should re-send the request to the new URI**. A response code of 301 indicate the resource has moved, and cannot be used to fulfill the request, so there's really no fallback position.

1. **Caching REST API Response**

[Caching](https://tools.ietf.org/html/rfc7234) is the ability to store copies of **frequently accessed data** in several places along the request-response path. When a consumer requests a resource representation, the request goes through a cache or a **series of caches** (**local cache, proxy cache, or reverse proxy**) toward the service hosting the resource. If any of the caches along the request path has a **fresh copy** of the requested representation, it uses that copy to satisfy the request. If none of the caches can satisfy the request, the request travels to the service (or **origin server** as it is formally known).

By using **HTTP headers**, **an origin server indicates whether a response can be cached** and**, if so, by whom, and for how long**. Caches along the response path can take a copy of a response, but only if the caching metadata allows them to do so.

Optimizing the network using caching improves the overall quality-of-service in the following ways:

* **Reduce bandwidth : Total number of users supported**
* **Reduce latency: speed of response**
* **Reduce load on servers:**
* **Hide network failures**

Being [cacheable](https://restfulapi.net/rest-architectural-constraints/#cacheable) is one of architectural constraints of REST.

**GET requests should be cachable by default** – until special condition arises. Usually, browsers treat all GET requests cacheable.

**POST** requests are **not cacheable by default** but can be made cacheable if either an Expires header or a Cache-Control header with a directive, to explicitly allows caching, is added to the response.

**Responses** to **PUT** and **DELETE** requests are **not cacheable** at all.

There are **two** main **HTTP response headers** that we can use to **control caching behavior**:

#### **Expires**

The Expires HTTP header specifies **an absolute expiry time** for a cached representation. Beyond that time, a cached representation is considered **stale** and must be re-validated with the origin server. To indicate that a representation never expires, a service can include a time up to one year in the future.

**Expires**: Fri, 20 May 2016 19:20:49 IST

#### **Cache-Control**

The header value comprises one or more comma-separated [**directives**](https://tools.ietf.org/html/rfc7234#page-24). These directives determine whether a response is cacheable, and if so, by whom, and for how long e.g. max-age or s-maxage directives.

**Cache**-**Control**: **max-age**=3600

Cacheable responses (whether to a GET or to a POST request) should also include a **validator** — either an **ETag** or a **Last-Modified** header.

#### **ETag**

An ETag value is an opaque string token that a server associates with a resource to uniquely identify the state of the resource over its lifetime. When the resource changes, the ETag changes accordingly.

**ETag**: "abcd1234567n34jv"

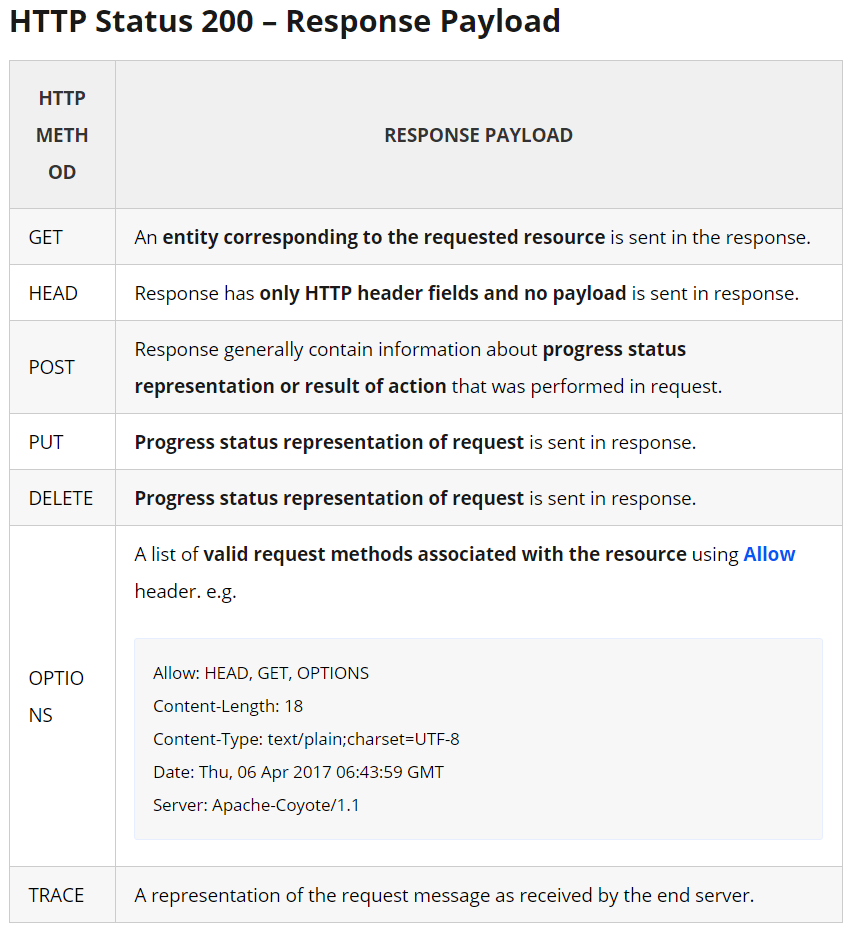
#### **Last-Modified**

Whereas a response’s Date header indicates when the response was generated, the Last-Modified header indicates when the associated resource last changed. The Last-Modified value cannot be less than the Date value.

**Last-Modified**: Fri, 10 May 2016 09:17:49 IST

1. **HTTP Status Code**
2. **HTTP Status 200 (OK)**

The **HTTP Status 200** (OK) status code indicates that the **request has been processed successfully** on server. The **response payload** depends on HTTP method which was selected for request.



## Important Facts

* A HTTP **200 response always has a payload**, though an origin server MAY generate a payload body of zero length or empty payload.
* If server does not want to send any payload in response, then it should send **HTTP status 204** (No Content) instead.
* By default, 200 (OK) response (header and payload) are cacheable. If caching needs to be overridden then response must include cache respective [cache headers](https://restfulapi.net/caching/).

1. **HTTP Status 201 (Created)**

HTTP Status 201 indicates that as a result of **HTTP POST** request, **one or more new resources have been successfully created** on server.

HTTP Headers

* The **response** may contain **URI** in **Location header** field in HTTP headers list, which can have reference to the newly created resource.

HTTP **POST response body: payload**

* Also, response payload also may 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.
* Or confirmation info about the Post action
* Or URI of the resource if it is available

*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.*

1. **HTTP Status 202 (Accepted)**

HTTP Status 202 indicates that the request has been accepted for processing, but the processing has not been completed. This status code is useful when the actual operation is asynchronous in nature.

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 describe **the request’s current status** and point to (or embed) a **status monitor** that can provide the user with (or without) an estimate of when the request will be fulfilled.

## HTTP Status 202 (Accepted) – Example

If you submitted a long-running asynchronous job to a REST API then API can return the result

like this:

|  |
| --- |
| HTTP STATUS 202 (Accepted)    {      "task": {          "href": "/api/company/job-management/jobs/2130040",          "id": "2130040"      }  } |

Now user agent can send **HTTP GET** request to URI /api/company/job-management/jobs/2130040 periodically for completion status of the job. **The response of above API** will inform the current status of actual scheduled operation.

#### Job Not Started

|  |
| --- |
| {     "job" : {       "@uri" : "/api/company/job-management/jobs/2130040" ,       "id" : "2130040",       "name" : "Update Resource",       "job-state" : "SCHEDULED",       "job-status" : "UNDETERMINED",       "percent-complete" : "0",       "scheduled-start-time" : "01-01-2013 10:50:45 PM GMT",       "start-time" : "",       "end-time" : "",       "owner" : "Admin",       "summary" : "random text"     }  } |

#### Job Started

|  |
| --- |
| {     "job" : {       "@uri" : "/api/company/job-management/jobs/2130040" ,       "id" : "2130040",       "name" : "Update Resource",       "job-state" : "STARTED",       "job-status" : "INPROGRESS",       "percent-complete" : "30",       "scheduled-start-time" : "01-01-2013 10:50:45 PM GMT",       "start-time" : "01-01-2013 10:50:55 PM GMT",       "end-time" : "",       "owner" : "Admin",       "summary" : "random text"     }  } |

#### Job Completed

|  |
| --- |
| {     "job" : {       "@uri" : "/api/company/job-management/jobs/2130040" ,       "id" : "2130040",       "name" : "Update Resource",       "job-state" : "COMPLETED",       "job-status" : "SUCCESS",       "percent-complete" : "100",       "scheduled-start-time" : "01-01-2013 10:50:45 PM GMT",       "start-time" : "01-01-2013 10:50:55 PM GMT",       "end-time" : ""01-01-2013 10:52:18 PM GMT",       "owner" : "Admin",       "summary" : "random text"     }  } |

# **HTTP Methods**

## **The History of HTTP Verbs**

When the [HTTP/1.1](http://tools.ietf.org/html/rfc2616) specification was first released in 1999, it established four of the five main HTTP verbs we still use today – GET, POST, PUT, and DELETE.

But it wasn’t until 2010 that [RFC 5789](http://tools.ietf.org/html/rfc5789) introduced the PATCH verb as a way of partially updating a resource. So before we even start to think about PATCH, what’s the difference between PUT and POST? A POST is used when you want to both create a resource AND have the server create a URI (Uniform Resource Identifier) for said resource.

For instance, if you wanted to create a new article, you would need to perform a POST to /articles and the article would be created and given a URI (for example, /articles/1234.)

But how do you create a resource via PUT? When using PUT, we expect to already KNOW the URI of the resource. So if I perform a PUT to /articles/1234 and it does not already exist, it will create that resource for me at that URI. I can also perform a PUT to /articles/1234 if it DOES exist, and it will update the resource.

It may seem like a minor difference, but it’s an important distinction. The HTTP specification considers PUT to be “idempotent,” while POST is not.

REST APIs enable you to develop any kind of web application having all possible CRUD (create, retrieve, update, delete) operations. [REST guidelines](https://restfulapi.net/rest-architectural-constraints/) suggest using a specific HTTP method on a particular type of call made to the server (though technically it is possible to violate this guideline, yet it is highly discouraged).

Table of Contents

[HTTP GET](https://restfulapi.net/http-methods/#get)

[HTTP POST](https://restfulapi.net/http-methods/#post)

[HTTP PUT](https://restfulapi.net/http-methods/#put)

[HTTP DELETE](https://restfulapi.net/http-methods/#delete)

[HTTP PATCH](https://restfulapi.net/http-methods/#patch)

## 

## **HTTP GET**

Use GET requests **to retrieve resource representation/information only** – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be **safe methods**. Additionally, GET APIs should be **idempotent**, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

For any given HTTP GET API, if the resource is found on the server, then it must return HTTP response code **200 (OK)** – along with the response body, which is usually either XML or JSON content (due to their platform-independent nature).

In case resource is NOT found on server then it must return HTTP response code **404 (NOT FOUND)**. Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code **400 (BAD REQUEST)**

#### Example request URIs

* HTTP GET http://www.appdomain.com/users
* HTTP GET http://www.appdomain.com/users?size=20&page=5
* HTTP GET http://www.appdomain.com/users/123
* HTTP GET <http://www.appdomain.com/users/123/address>

1. **HTTP POST**

When talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.

Ideally, if a resource has been created on the origin server, the **response** SHOULD be HTTP response code **201 (Created)** and **contain an entity** which describes the status of the request and refers to the **new resource**, and a [**Location**](https://en.wikipedia.org/wiki/HTTP_location)**header**.

Many times, the action performed by the POST method **might not result in a resource that can be identified by a URI**. In this case, either HTTP response code **200 (OK)** or **204 (No Content)** is the appropriate response status.

Now let us take a look at a real example.

When I do POST to /users with the data {username: 'skwee357', email: 'skwee357@domain.com'} **and** the server is capable of creating a resource,

it will respond with **201** and **resource location** (lets assume /users/1) and any next call to GET /users/1 will return {id: 1, username: 'skwee357', email: 'skwee357@domain.com'}.

Responses to this method are **not cacheable**, unless the response includes appropriate [Cache-Control](https://en.wikipedia.org/wiki/Web_cache#Cache_control) or [Expires](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html) header fields.

Please note that POST is **neither safe nor idempotent**, and invoking two identical POST requests will result in two different resources containing the same information (except resource ids).

#### Example request URIs

* HTTP POST http://www.appdomain.com/users
* HTTP POST <http://www.appdomain.com/users/123/accounts>

This allows the client to create resources without knowing the URI for the new resource. For example, we can send a POST request to */projects* to create a new project. The server can now create the project as a new subordinate of */project*, for example: */projects/123*. So when using POST for resource creation the server can decide the URI (and typically the ID) of the newly created resources.

* **When the server created a resource, it should respond with the 201 (Created) status code and a *Location* header that points to the newly created resource.**
* For example:

**Request:**

* POST /projects HTTP/1.1
* Content-Type: application/json
* {
* "name": "my cool project",
* ...
* }

**Response:**

* **HTTP/1.1 201 Created**
* **Location: https://cool.api.com/projects/123**

1. **HTTP PUT**

Use PUT APIs primarily **to update existing resource** (**if the resource does not exist**, then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and **if an existing resource is modified**, either the **200 (OK)** or **204 (No Content)** response codes SHOULD be sent to indicate successful completion of the request.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are **not cacheable**.

***The difference between the POST and PUT APIs can be observed in request URIs****. POST requests are made on resource collections, whereas PUT requests are made on a single resource.*

#### Example request URIs

* HTTP PUT http://www.appdomain.com/users/123
* HTTP PUT <http://www.appdomain.com/users/123/accounts/456>

## **Lost update problem**

HTTP 201 response MAY contain an ETag response header field indicating the current value of the entity tag for the requested variant just created. ETag header field can be used in later conditional requests to prevent the “lost update” problem.

Lost update problem happens when **multiple people edit a resource** without knowledge of each others’ changes. In this scenario, the last person to update a resource “wins”, and previous updates are lost. **ETag**s can be used in combination with the **If-Match** header to let the server decide if a resource should be updated. If ETag does not match then server informs the client via a **412 (Precondition Failed)** response.

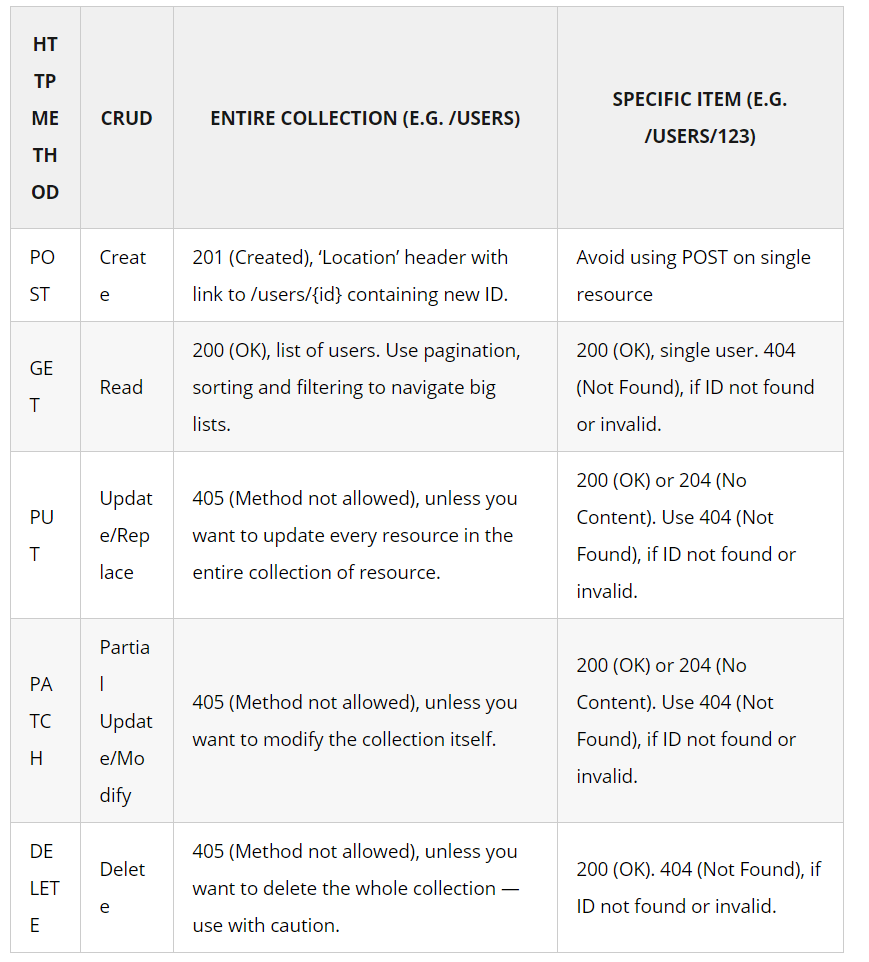
1. **HTTP PATCH (Partial Updates)**

HTTP PATCH requests are **to make partial update on a resource**. If you see PUT requests also modify a resource entity, so to make more clear – **PATCH method is the correct choice for partially updating an existing resource**, and **PUT should only be used if you’re replacing a resource in its entirety**.

PATCH is a relatively new verb (RFC introduced in March 2010), and it comes to solve the problem of "patching" or modifying a set of fields. Before PATCH was introduced, everybody used PUT to update resources.

Please note that there are some challenges if you decide to use PATCH APIs in your application:

* **Support** for PATCH in browsers, servers, and web application frameworks **is not universal**. IE8, PHP, Tomcat, Django, and lots of other software has missing or broken support for it.

****

1. **Put vs Patch**

* **Put** is idempotent. Patch loses the idempotence and its benefits such as **reliable caching** of responses on the network and **reliable updates of resources from retries** when the original request fails. PUT requests are particularly useful for major updates.

**Responses** to **PUT requests** are not cacheable. If a PUT request finds a response in a cache infrastructure, that response (cache entry) should be treated as stale

A **PATCH** request on the other hand, is used to make changes to part of the resource at a location. It is used to make minor updates to resources and it’s not required to be idempotent.

Since PATCH is not idempotent, **failed requests are not automatically re-attempted on the network**. Also, if a PATCH request is made to a non-existent url e.g attempting to replace the front door of a non-existent building, it should simply fail without creating a new resource unlike PUT, which would create a new one using the payload.

* In RESTful API designs, PUT requests are generally used to **add** or replace an entire resource, whereas a PATCH should be just used to update an existing resource. It is implementation dependent if PUT will create a new resource if the resource needed to be replaced doesNOT exist.
* Often **PATCH requests use the same format** as the resource that should be updated and just omit the fields that should not change. **However, it does not have to be this way**. It is also fine to use a separate patch format, which describes how the resource should be modified.
* PATCH is neither [safe nor idempotent](https://www.mscharhag.com/api-design/http-idempotent-safe).
* Maybe you are wondering in which situations a partial resource update is not idempotent. A simple example here is the addition of an item to an existing list resource, like adding a product to a shopping cart. Multiple (partial) update requests might add the product multiple times to the shopping cart.

Before we start, let's quickly check why partial updates (PATCH) can be useful:

* Simplicity - If a client only wants to update a single field, a partial update request can be simpler to implement.
* Bandwidth - If your resource representations are quite large, partial updates can reduce the amount of bandwidth required.
* Lost updates - Resource replacements with PUT can be susceptible for the lost update problem. While partial updates do not solve this problem, they can help reducing the number of possible conflicts.

1. **SAFE Method**

As per HTTP specification, the **GET and HEAD methods should be used only for retrieval of resource representations** – and they do not update/delete the resource on the server. Both methods are said to be considered “**safe**“.

This allows user agents to represent other methods, such as **POST, PUT and DELETE**, in a unique way so that the user is made aware of the fact that a possibly unsafe action is being requested – and they can **update/delete the resource on server** and so should be used carefully.

### Idempotent Methods

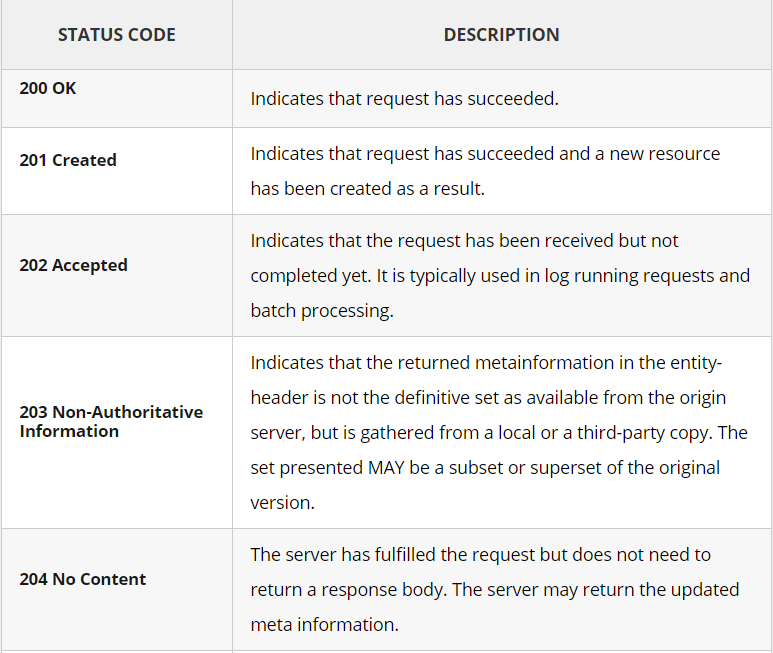
The term idempotent is used more comprehensively to describe an **operation that will produce the same results if executed once or multiple times**. Idempotence is a handy property in many situations, as it means that an operation can be repeated or retried as often as necessary without causing unintended effects. With **non-idempotent (POST, PATCH)** operations, the algorithm may have to keep track of whether the operation was already performed or not.

In HTTP specification, The methods **GET, HEAD, PUT and DELETE are declared idempotent methods**. Other methods OPTIONS and TRACE SHOULD NOT have side effects, so both are also inherently idempotent.

1. **Status Code**

[**https://httpstatuses.com/**](https://httpstatuses.com/)

**https://evertpot.com/http/412-precondition-failed**

****

#### [**200 (OK)**](https://restfulapi.net/http-status-200-ok/)

GET

It indicates that the REST API successfully carried out whatever action the client requested and that no more specific code in the 2xx series is appropriate.

Unlike the 204 status code, a 200 response should include a response body. 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.

#### [**201 (Created)**](https://restfulapi.net/http-status-201-created/)

POST

A REST API responds with the 201 status code whenever a resource is created inside a collection. There may also be times when a new resource is created as a result of some controller action, in which case 201 would also be an appropriate response.

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 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 a 202 (Accepted) response instead.

#### [**202 (Accepted)**](https://restfulapi.net/http-status-202-accepted/)

A 202 response is typically used for actions that take a long while to process. It indicates that the request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, or even maybe disallowed when processing occurs.

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 (job queue location) or some estimate of when the user can expect the request to be fulfilled.

#### 203 Non-Authoritative Information

#### [**204 (No Content)**](https://restfulapi.net/http-status-204-no-content/)

The 204 status code is usually sent out in response to a **PUT, PATCH, POST, or DELETE** request when the REST API declines to send back any status message or representation in the response message’s body.

An API may also send 204 in conjunction with a GET request to indicate that the requested resource exists, but has no state representation to include in the body.

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. However, any new or updated metainformation SHOULD be applied to the document currently in the user agent’s dynamic view.

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

#### [**301 (Moved Permanently)**](https://restfulapi.net/http-status-301-moved-permanently/)

**GET**

The 301 status code indicates that the REST API’s resource model has been significantly redesigned, and a **new permanent URI has been assigned** to the client’s requested resource. The REST API should specify the new URI in the response’s **Location header**, and all future requests should be directed to the given URI.

You will hardly use this response code in your API as you can always use the **API versioning** for new API while retaining the old one.

#### **304 (Not Modified)**

A conditional **GET** or **HEAD** request has been received and would have resulted in a [**200 OK**](https://httpstatuses.com/200) response if it were not for the fact that the condition evaluated to false.

In other words, there is no need for the server to transfer a representation of the target resource because the request indicates that **the client, which made the request conditional**, already has a valid representation; the server is therefore **redirecting the client to make use of that stored representation** as if it were the payload of a [**200 OK**](https://httpstatuses.com/200) response.

The server generating a 304 response MUST generate any of the following header fields that would have been sent in a [**200 OK**](https://httpstatuses.com/200) response to the same request: Cache-Control, Content-Location, Date, ETag, Expires, and Vary.

Since the goal of a 304 response is to minimize information transfer when the recipient already has one or more cached representations, a sender SHOULD NOT generate representation metadata other than the above listed fields unless said metadata exists for the purpose of guiding cache updates (e.g., Last-Modified might be useful if the response does not have an ETag field).

Requirements on a cache that receives a 304 response are defined in [**Section 4.3.4 of RFC7234**](https://tools.ietf.org/html/rfc7234#section-4.3.4). If the conditional request originated with an outbound client, such as a user agent with its own cache sending a conditional GET to a shared proxy, then the proxy SHOULD forward the 304 response to that client.

A 304 response cannot contain a message-body; it is always terminated by the first empty line after the header fields.

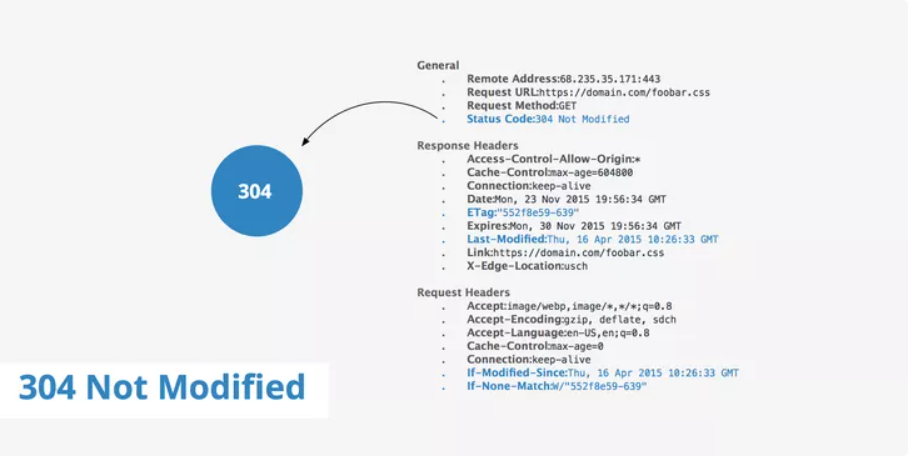
This status code is similar to 204 (“No Content”) in that **the response body must be empty**. The critical distinction is that 204 is used when there is nothing to send in the body, whereas 304 is used when the resource has not been modified since the version specified by

the request headers **If-Modified-Since** or **If-None-Match**.

* **If-Match**
* **If-None-Match**
* If-Modified-Since
* If-Unmodified-Since
* If-Range

In such a case, there is no need to retransmit the resource since the client still has a previously-downloaded copy.

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.



## What does 304 Not Modified mean?[#](https://www.keycdn.com/support/304-not-modified#what-does-304-not-modified-mean)

304 Not Modified is an HTTP status code that is returned to the client when the cached copy of a particular file is up to date with the server. When a client such as a browser stores something in cache, it also keeps the Last-Modified header sent from the server. This header provides the browser with the **date and time** of when the file was last modified.

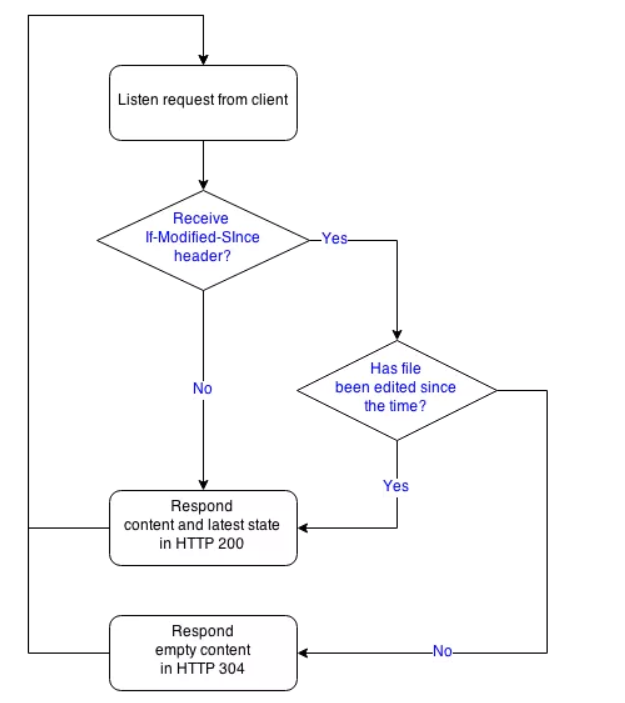
When a request is made from the browser for the same file again, the [If-Modified-Since](https://www.keycdn.com/support/if-modified-since-http-header) header is sent to the server. If this value and the server's ETag value for that file are both the same, the server sends back the 304 Not Modified HTTP status.

## Using If-Modified-Since vs If-None-Match[#](https://www.keycdn.com/support/304-not-modified#using-if-modified-since-vs-if-none-match)

Using the If-Modified-Since header is not the only way to determine whether the server should respond with a 304 Not Modified. There is in fact, another header that can be used, which is the If-None-Match header. Both of these options are further discussed in the section below.

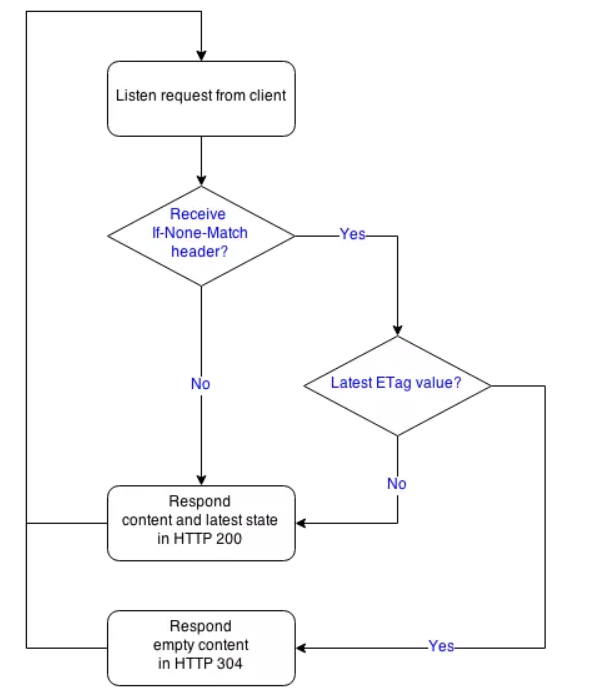
### If-Modified-Since[#](https://www.keycdn.com/support/304-not-modified#if-modified-since)

In order to visualize the flow of information from the browser to the server and back, consider the following workflow.

Once the server receives a request from a client, it checks to see if the If-Modified-Since header is present. If no, the server will deliver the latest content to the client as this is the first time it is being requested. On the other hand, if the If-Modified-Since header is present, the server will check to see if the file has been edited since the last time the browser accessed the file. If the answer is yes, the server returns a **new copy** to the client, if no, the server responds with a 304 Not Modified and the browser can retrieve the content from its cache.

### If-None-Match[#](https://www.keycdn.com/support/304-not-modified#if-none-match)

The workflow of using If-None-Match is quite similar to If-Modified-Since as can be seen by the diagram below.



Using the If-None-Match option to determine if a file has changed involves the creation of a unique identifier called an ETag. An ETag usually consists of a string of text and numbers generated by the server, resembling something similar to: flask-1448298997.74-7011-3318025237. Once the client has requested a file, it will receive an ETag which the browser will use in conjunction with the If-None-Match header for subsequent requests, similar to: If-None-Match: "flask-1448298997.74-7011-3318025237".

Since the server generates a new and maintains the previous ETag value **each time the file is changed**, it will check to see if the browser's ETag matches its own. If not, the server sends a new copy of the file and if yes, the 304 Not Modified header is returned to the client.

## Summary[#](https://www.keycdn.com/support/304-not-modified#summary)

Both options mentioned above are suitable ways to determine whether or not the server should return a 304 Not Modified status code. The If-Modified-Since header is often used by CDNs to more efficiently deliver static content such as CSS, JavaScript, or images. Using the If-None-Match header in conjunction with ETags can serve more useful when the content is dynamic and we would like to keep it in temporary memory.

The 304 Not Modified status code is a fairly simple concept that harnesses the ability to **save both resources and time**. Each time a file must be delivered directly from the origin server, this means that the user must wait for the file to download which in turn uses bandwidth on the server's end. With the use of the 304 Not Modified header, browsers and servers are able to efficiently communicate what needs to be downloaded again and what can be delivered from cache.

#### **307 (Temporary Redirect)**

A 307 response indicates that the **REST API is not going to process the client’s request**. Instead, the **client should resubmit the request to the URI specified by the response message’s Location header**. However, **future requests should still use the original URI**.

A REST API can use this status code to assign a temporary URI to the client’s requested resource. For example, a 307 response can be used to shift a client request over to another host.

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 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.

**308 PERMANENT REDIRECT**

The target resource has been assigned a new permanent URI and any future references to this resource ought to use one of the enclosed URIs.

Clients with link editing capabilities ought to automatically re-link references to the effective request URI[**1**](https://httpstatuses.com/308#ref-1) to one or more of the new references sent by the server, where possible.

The server SHOULD generate a **Location** header field[**2**](https://httpstatuses.com/308#ref-2) in the response **containing a preferred URI reference for the new permanent URI**. The user agent MAY use the Location field value for **automatic redirection**. The server's response payload usually contains a short hypertext note with a hyperlink to the new URI(s).

**A 308 response is cacheable by default**; i.e., unless otherwise indicated by the method definition or explicit cache controls[**3**](https://httpstatuses.com/308#ref-3).

Note: This status code is similar to [**301 Moved Permanently**](https://httpstatuses.com/301), except that it does not allow changing the request method from POST to GET.

**4xx Client Error**

**400 (Bad Request)**

The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

400 is the generic client-side error status, **used when no other 4xx error code is appropriate**. Errors can be like malformed request **syntax**, **invalid request message parameters**, or deceptive request routing etc.

The client SHOULD NOT repeat the request without modifications.

**401 (Unauthorized)**

The request has not been applied because it lacks valid authentication credentials for the target resource.

A 401 error response indicates that the client tried to operate on a protected resource without providing the proper **authorization**. It may have provided **the wrong credentials or none at all**. The **response** must include a **WWW-Authenticate header** field containing a challenge applicable to the requested resource.

The client MAY **repeat the request with a suitable Authorization header field**. 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.

**403 (Forbidden)**

A 403 error response indicates that the client’s request is formed correctly, but the REST API refuses to honor it, i.e., **the user does not have the necessary permissions for the resource**. A 403 response is not a case of insufficient client credentials; that would be 401 (“Unauthorized”).

Authentication will not help, and the request SHOULD NOT be repeated. Unlike a 401 Unauthorized response, authenticating will make no difference.

The server understood the request but refuses to authorize it.

A server that wishes to make public why the request has been forbidden can describe that reason in the response payload (if any).

If authentication credentials were provided in the request, the server considers them insufficient to grant access. The client SHOULD NOT automatically repeat the request with the same credentials. The client MAY repeat the request with new or different credentials. However, a request might be forbidden for reasons unrelated to the credentials.

An origin server that wishes to "hide" the current existence of a forbidden target resource MAY instead respond with a status code of [**404 Not Found**](https://httpstatuses.com/404).

**404 (Not Found)**

The 404 error status code indicates that the REST API can’t map the client’s URI to a resource but may be available in the future. Subsequent requests by the client are permissible.

The origin server did not find a current representation for the target resource or is not willing to disclose that one exists.

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.

**405 (Method Not Allowed)**

The API responds with a 405 error to indicate that the client tried to use an HTTP method that the **resource does not allow**. For instance, a read-only resource could support only GET and HEAD, while a controller resource might allow GET and POST, but not PUT or DELETE.

A 405 **response** must include the **Allow header**, which lists the HTTP methods that the resource supports. For example:

Allow: GET, POST

**406 (Not Acceptable)**

**406 Not Acceptable** client error response code indicates that **the server cannot produce a response** matching the list of acceptable values defined in the request's proactive [content negotiation](https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation) headers, and that the server is unwilling to supply a default representation. This is **content-type** related.

Proactive content negotiation headers include:

* [Accept](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept)
* [Accept-Charset](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Charset)
* [Accept-Encoding](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Encoding)
* [Accept-Language](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept-Language)

**412 Precondition Failed**

In HTTP it’s possible to do conditional requests. These are requests that only execute if the right conditions are met.

For GET requests, this might be done to only retrieve the resource if it has changed. For those cases [304 Not Modified](https://evertpot.com/http/304-not-modified) is returned.

For other cases, 412 Precondition Failed is returned.

## Examples

This client only wants the PUT request to succeed, if it didn’t already exist:

PUT /foo/new-article.md HTTP/1.1

Content-Type: text/markdown

If-None-Match: \*

This request is an update, and it should only succeed if the article hasn’t changed since last time.

PUT /foo/old-article.md HTTP/1.1

If-Match: "1345-12315"

Content-Type: text/markdown

If the condition didn’t pass, it returns:

HTTP/1.1 412 Precondition Failed

Content-Type: text/plain

The article you're tring to update has changed since you last seen it.

One great advantage of this is that **prevents lost updates**, due to multiple people writing to the same resource. This is also known as the ‘lost update’ problem.

#### **415 (Unsupported Media Type)**

The 415 error response indicates that the **server is not able to process** the client’s supplied media type, as indicated by the **Content-Type** request header. For example, a client request including data formatted as application/xml will receive a 415 response if the API is only willing to process data formatted as application/json.

For example, the client uploads an image as image/svg+xml, but the server requires that images use a different format.

#### **406 (Not Acceptable) vs 415 (Unsupported Media Type)**

* **406**: Server can’t provide resource in the media type requested/accepted by client.

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.

* **415**: Server does not understand or can NOT process the media type sent by client request.

The server is refusing to service the request because the entity of the request is in a format (**Content-Type** )not supported by the requested resource for the requested method

#### **500 (Internal Server Error)**

500 is the generic REST API error response. Most web frameworks automatically respond with this response status code **whenever they execute some request handler code that raises an exception**.

A 500 error is never the client’s fault, and therefore, it is reasonable for the client to retry the same request that triggered this response and hope to get a different response.

The API response is the generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

#### **501 (Not Implemented)**

The server either does not recognize the request method, or it cannot fulfill the request. Usually, this implies future availability (e.g., a new feature of a web-service API).

**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.

A 501 response is cacheable by default; i.e., unless otherwise indicated by the method definition or explicit cache controls.

# **502 Bad Gateway**

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

HTTP is a protocol that is implemented by servers and clients, but there is a third category: **proxies**.

When a system is acting as a proxy for a different server, and that server is misbehaving or doing something unexpected, **the proxy can return** [502 Bad Gateway](https://tools.ietf.org/html/rfc7231#section-6.6.3) to tell a client that the proxy is working fine, but there was something wrong with the ‘origin’ server instead.

A specific example of this could be a CDN in front on a web server, and the web-server is misconfigured and responding with incorrect HTTP responses.

This status should not be used if the origin server just returned a valid HTTP error itself, because these should generally just be forwarded by the proxy (mostly) unaltered.

## Example

HTTP/1.1 502 Bad Gateway

Content-Type text/plain

We made a HTTP request to an origin server, but we got a Gopher response back.

**503 (Service Unavailable Error)**

indicating that a server is temporarily unable to handle the request. **This may be due to the server being overloaded or down for maintenance**. This particular response code differs from a code like the [500 Internal Server Error](https://airbrake.io/blog/http-errors/500-internal-server-error) we explored some time ago. While a 500 Internal Server Error indicates an issue preventing the server from handling the request entirely, a 503 Service Unavailable Error is an indication that the server is **still functioning properly**, since it’s able to process the request and has opted to return the 503 response code.

The server is currently unable to handle the request due to a temporary overload or scheduled maintenance, which will likely be alleviated after some delay.

The server MAY send a **Retry-After** header field[**1**](https://httpstatuses.com/503#ref-1) to suggest an appropriate amount of time for the client to wait before retrying the request.

## **Troubleshooting on the Server-Side**

Here are some additional tips to help you troubleshoot what might be causing the **503 Service Unavailable**to appear on the server-side of things:

* Reboot the Server – If you or an administrator have the ability to do so, one of the simplest solutions is often to restart the web server hosting the application. If your application is spread over multiple servers, make sure all are rebooted in the proper manner so the system is brought back online as normal. A 503 Service Unavailable code could be a result of a bottleneck somewhere in the server chain that hosts your application, so a simple reboot could refresh everything and get you back up and running.
* Check for Unexpected Maintenance – You may not realize it, but your server and/or application may be configured to go down for maintenance automatically. Many modern content management systems like WordPress will automatically download and install updates to their base software without any intervention on your behalf. The web server could be issuing 503 Service Unavailable Errors during this period, particularly if your application resides on a slower server or a shared host. If you’re able to access the administration settings of your application/server, check the configuration options for automatic maintenance scheduling or the like, disabling such options if you’d rather have direct control over that process (just don’t forget to upgrade to newer versions fairly regularly, as they typically include critical security fixes).
* Server Connectivity Issues – While it may sound simple, it’s entirely possible that a 503 Service Unavailable Error simply indicates that a server somewhere in the chain is down or unreachable for whatever reason. Most modern applications don’t reside on a single server, but may, instead, be spread over multiple systems, or even rely on many third-party services to function. If any one of these servers are down for maintenance or otherwise inaccessible, this could result in an error that appears to be from your own application.
* Improper Firewall Configuration – A firewall is a basic security device that monitors network traffic and acts as a gatekeeper, deciding which traffic is safe and which could be malicious. In most cases, all potentially harmful traffic is stopped (and may be logged for network admin use). In some situations, it’s entirely possible for a firewall configured somewhere on the network in which your application is running to be preventing some form of critical traffic from getting through. This is particularly true for applications that rely on content delivery networks (CDNs), which act as a third-party host for “heavy” content like images or videos, hosting that content on behalf of your application, so your application can maintain its speed and efficiency. However, automatic firewall services can sometimes perform false positives, mistaking perfectly safe and valid content from CDNs or elsewhere as malicious, thereby shutting off that stream of content in an instant, which could lead to a 503 Service Unavailable Error.
* Check the Logs – Nearly every web application will keep some form of server-side logs. Application logs are typically the history of what the application did, such as which pages were requested, which servers it connected to, which database results it provides, and so forth. Server logs are related to the actual hardware that is running the application, and will often provide details about the health and status of all connected services, or even just the server itself. Google “logs [PLATFORM\_NAME]” if you’re using a CMS, or “logs [PROGRAMMING\_LANGUAGE]” and “logs [OPERATING\_SYSTEM]” if you’re running a custom application, to get more information on finding the logs in question.
* Application Code or Script Bugs – If all else fails, it may be that a problem in some custom code within your application is causing the issue. Try to diagnose where the issue may be coming from through manually debugging your application, along with parsing through application and server logs. Ideally, make a copy of the entire application to a local development machine and perform a step-by-step debug process, which will allow you to recreate the exact scenario in which the 503 Service Unavailable Error occurred and view the application code at the moment something goes wrong.

No matter what the cause, the appearance of a 503 Service Unavailable Error within your own web application is a strong indication that you may need an error management tool to help you automatically detect such errors in the future. The best of these tools can even alert you and your team immediately when an error occurs. Airbrake’s error monitoring software provides real-time [error monitoring](https://airbrake.io/) and automatic exception reporting for all your development projects. Airbrake’s state of the art web dashboard ensures you receive round-the-clock status updates on your application’s health and error rates. No matter what you’re working on, Airbrake easily integrates with all the most popular languages and frameworks. Plus, Airbrake makes it easy to customize exception parameters, while giving you complete control of the active error filter system, so you only gather the errors that matter most.

Check out [Airbrake’s error monitoring software](https://airbrake.io/new) today and see for yourself why so many of the world’s best engineering teams use Airbrake to revolutionize their exception handling practices!

1. **API Security**

There are multiple ways to secure a RESTful API e.g. [**basic auth**](https://howtodoinjava.com/resteasy/jax-rs-resteasy-basic-authentication-and-authorization-tutorial/), [**OAuth**](https://oauth.net/) etc.

but one thing is sure that RESTful APIs should be stateless – so request authentication/authorization should not depend on cookies or sessions. **Instead, each API request should come with some sort authentication credentials which must be validated on the server for every request.**

#### **Always Use HTTPS**

By always using **[SSL](https://www.digicert.com/ssl/)**, (Secure Sockets Layer) the authentication credentials can be simplified to a randomly generated access token. The token is delivered in the username field of **HTTP Basic Auth**. It’s relatively simple to use, and you get a lot of security features for free.

If you use [HTTP 2](https://http2.github.io/), to improve performance – you can even [send multiple requests over a single connection](https://en.wikipedia.org/wiki/HTTP_persistent_connection), that way you avoid the complete TCP and SSL handshake overhead on later requests.

#### **Use Password Hash**

Passwords must always be hashed to protect the system (or minimize the damage) even if it is compromised in some hacking attempts. There are many such [hashing algorithms](https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/) which can prove really effective for password security e.g. PBKDF2, bcrypt and scrypt algorithms.

#### **Never expose information on URLs**

Usernames, passwords**, session tokens**, and **API keys** should not appear in the URL, as this can be captured in web server logs, which makes them easily exploitable.

https://api.domain.com/user-management/users/{id}/someAction?apiKey=abcd123456789 //Very BAD !!

The above URL exposes the API key. So, never use this form of security.

#### Consider **OAuth**

Though [basic auth](https://en.wikipedia.org/wiki/Basic_access_authentication) is good enough for most of the APIs and if implemented correctly, it’s secure as well – yet you may want to consider [OAuth](https://tools.ietf.org/html/rfc6749) as well. The **OAuth 2.0 authorization framework** enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its behalf.

#### Consider Adding Timestamp in Request

Along with other request parameters, you may add a request timestamp as an HTTP custom header in API requests. The server will compare the current timestamp to the request timestamp and only accepts the request if it is within a reasonable timeframe (1-2 minutes, perhaps).

This will prevent very basic [replay attacks](https://en.wikipedia.org/wiki/Replay_attack) from people who are trying to [brute force](https://en.wikipedia.org/wiki/Brute-force_attack) your system without changing this timestamp.

#### **Input Parameter Validation**

Validate request parameters on the very first step, before it reaches to application logic. Put strong validation checks and reject the request immediately if validation fails. In API response, send relevant error messages and example of correct input format to improve user experience.

1. **API Versioning**

To manage this complexity, version your API. Versioning helps you iterate faster when the needed changes are identified.

*Change in an API is inevitable as your knowledge and experience of a system improve. Managing the impact of this change can be quite a challenge when it threatens to break existing client integration.*

## **When to Version**

APIs only need to be **up-versioned** when a breaking change is made. **Breaking changes** include:

* a change in the **format of the response data** for one or more calls
* a change in the **request or response type** (i.e. changing an integer to a float)
* removing any part of the API.

**Breaking changes** should always result in a change to the **major version number** for an API or content [response type](https://www.iana.org/assignments/media-types/media-types.xhtml).

**Non-breaking changes**,

such as **adding new endpoints or new response parameters**, do not require a change to the major version number. However, it can be helpful to track the **minor versions** of APIs when changes are made to support customers who may be receiving cached versions of data or may be experiencing other API issues.

## How to Version

REST doesn’t provide for any specific versioning guidelines but the more commonly used approaches fall into three categories:

### URI Versioning

Using the URI is the most straightforward approach (and most commonly used as well) though it does violate the principle that a URI should refer to a unique resource. You are also guaranteed to break client integration when a version is updated.

e.g.

http://api.example.com/v1

http://apiv1.example.com

The version need not be numeric, nor specified using the “v[x]” syntax. Alternatives include dates, project names, seasons or other identifiers that are meaningful enough to the team producing the APIs and flexible enough to change as the versions change.

### Versioning using Custom Request Header

A custom header (e.g. Accept-version) allows you to preserve your URIs between versions though it is effectively a duplicate of the content negotiation behavior implemented by the existing Accept header.

e.g.

Accept-version: v1

Accept-version: v2

### Versioning using Accept header

Content negotiation may let you preserve a clean set of URLs but you still have to deal with the complexity of serving different versions of content somewhere. This burden tends to be moved up the stack to your API controllers which become responsible for figuring out which version of a resource to send. The end result tends to be a more complex API as clients have to know which headers to specify before requesting a resource.

e.g.

Accept: application/vnd.example.v1+json

Accept: application/vnd.example+json;version=1.0

In the real world, an API is never going to be completely stable. So it’s important how this change is managed. A well documented and gradual deprecation of API can be an acceptable practice for most of the APIs.

1. **API Naming**

# REST Resource Naming Guide

In REST, primary data representation is called **Resource**. Having a strong and consistent REST resource naming strategy – will prove one of the best design decisions in the long term.

The key abstraction of information in REST is a **resource**. **Any information** that can be named can be a resource: a **document** or **image**, **a temporal service** (e.g. “today’s weather in Los Angeles”), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author’s hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

A **resource can be a singleton or a collection**. For example, “customers” is a collection resource and “customer” is a singleton resource (in a banking domain). We can identify “customers” collection resource using the URI “/customers”. We can identify a single “customer” resource using the URI “/customers/{customerId}”.

A **resource may contain sub-collection resources** also. For example, sub-collection resource “accounts” of a particular “customer” can be identified using the URN “/customers/{customerId}/accounts” (in a banking domain). Similarly, a singleton resource “account” inside the sub-collection resource “accounts” can be identified as follows: “/customers/{customerId}/accounts/{accountId}”.

REST APIs use [Uniform Resource Identifiers](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier) (URIs) to address resources. REST API designers should create URIs that convey a REST API’s resource model to its potential client developers. When resources are named well, an API is intuitive and easy to use. If done poorly, that same API can feel difficult to use and understand.

The constraint of a **uniform interface** is partially addressed by the combination of URIs and HTTP verbs and using them in line with the standards and conventions.

### Use nouns to represent resources

RESTful URI should refer to a resource that is a thing (noun) instead of referring to an action (verb) because nouns have properties which verbs do not have – similar to resources have attributes. Some examples of a resource are:

* **Users of the system**
* **User Accounts**
* **Network Devices etc**.

and their resource URIs can be designed as below:

http://api.example.com/device-management/managed-devices

http://api.example.com/device-management/managed-devices/{device-id}

http://api.example.com/user-management/users/

http://api.example.com/user-management/users/{id}

For more clarity, let’s divide the **resource archetypes** into four categories (document, collection, store and controller) and then **you should always target to put a resource into one archetype and then use it’s naming convention consistently**. For uniformity’s sake, resist the temptation to design resources that are hybrids of more than one archetype.

#### **document**

A document resource is a singular concept that is akin to an object instance or database record. In REST, you can view it as a single resource inside resource collection. A document’s state representation typically includes both fields with values and links to other related resources.

Use “singular” name to denote document resource archetype.

http://api.example.com/device-management/managed-devices/{device-id}

http://api.example.com/user-management/users/{id}

http://api.example.com/user-management/users/admin

#### **collection**

A collection resource is a server-managed directory of resources. Clients may propose new resources to be added to a collection. However, it is up to the collection to choose to create a new resource or not. A collection resource chooses what it wants to contain and also decides the URIs of each contained resource.

Use the “plural” name to denote the collection resource archetype.

http://api.example.com/device-management/managed-devices

http://api.example.com/user-management/users

http://api.example.com/user-management/users/{id}/accounts

#### **store**

A store is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. A store never generates new URIs. Instead, each stored resource has a URI. The URI was chosen by a client when it was initially put into the store.

Use “plural” name to denote store resource archetype.

http://api.example.com/song-management/users/{id}/playlists

#### **controller**

**A controller resource** models a **procedural concept**. Controller resources are like executable functions, with parameters and return values; inputs and outputs.

Use “verb” to denote controller archetype.

http://api.example.com/cart-management/users/{id}/cart/checkout

http://api.example.com/song-management/users/{id}/playlist/play

### Consistency is the key

Use consistent resource naming conventions and URI formatting for minimum ambiguily and maximum readability and maintainability. You may implement below design hints to achieve consistency:

#### Use forward slash (/) to indicate hierarchical relationships

The forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources. e.g.

http://api.example.com/device-management

http://api.example.com/device-management/managed-devices

http://api.example.com/device-management/managed-devices/{id}

http://api.example.com/device-management/managed-devices/{id}/scripts

http://api.example.com/device-management/managed-devices/{id}/scripts/{id}

#### Do not use trailing forward slash (/) in URIs

As the last character within a URI’s path, a forward slash (/) adds no semantic value and may cause confusion. It’s better to drop them completely.

http://api.example.com/device-management/managed-devices/

http://api.example.com/device-management/managed-devices /\*This is much better version\*/

#### **Use hyphens (-) to improve the readability of URIs**

To make your URIs easy for people to scan and interpret, use the hyphen (-) character to improve the readability of names in long path segments.

http://api.example.com/inventory-management/managed-entities/{id}/install-script-location //More readable

http://api.example.com/inventory-management/managedEntities/{id}/installScriptLocation //Less readable

#### Do not use underscores ( \_ )

It’s possible to use an underscore in place of a hyphen to be used as separator – But depending on the application’s font, it’s possible that the underscore (\_) character can either get partially obscured or completely hidden in some browsers or screens.

To avoid this confusion, use hyphens (-) instead of underscores ( \_ ).

http://api.example.com/inventory-management/managed-entities/{id}/install-script-location //More readable

http://api.example.com/inventory\_management/managed\_entities/{id}/install\_script\_location //More error prone

#### Use lowercase letters in URIs

When convenient, lowercase letters should be consistently preferred in URI paths.

[RFC 3986](https://www.ietf.org/rfc/rfc3986.txt) defines URIs as case-sensitive except for the scheme and host components. e.g.

http://api.example.org/my-folder/my-doc //1

HTTP://API.EXAMPLE.ORG/my-folder/my-doc //2

http://api.example.org/My-Folder/my-doc //3

In above examples, 1 and 2 are same but 3 is not as it uses **My-Folder** in capital letters.

#### **Do not use file extentions**

File extensions look bad and do not add any advantage. Removing them decreases the length of URIs as well. No reason to keep them.

Apart from above reason, if you want to highlight the media type of API using file extenstion then you should rely on the media type, as communicated through the Content-Type header, to determine how to process the body’s content.

http://api.example.com/device-management/managed-devices.xml /\*Do not use it\*/

http://api.example.com/device-management/managed-devices /\*This is correct URI\*/

### Never use CRUD function names in URIs

URIs should not be used to indicate that a CRUD function is performed. URIs should be used to uniquely identify resources and not any action upon them. HTTP request methods should be used to indicate which CRUD function is performed.

HTTP GET http://api.example.com/device-management/managed-devices //Get all devices

HTTP POST http://api.example.com/device-management/managed-devices //Create new Device

HTTP GET http://api.example.com/device-management/managed-devices/{id} //Get device for given Id

HTTP PUT http://api.example.com/device-management/managed-devices/{id} //Update device for given Id

HTTP DELETE http://api.example.com/device-management/managed-devices/{id} //Delete device for given Id

### Use query component to filter URI collection

Many times, you will come across requirements where you will need a collection of resources sorted, filtered or limited based on some certain resource attribute. For this, do not create new APIs – rather enable sorting, filtering and pagination capabilities in resource collection API and pass the input parameters as query parameters. e.g.

http://api.example.com/device-management/managed-devices

http://api.example.com/device-management/managed-devices?region=USA

http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ

http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ&sort=installation-date

Let’s say you want to define the end-points for a College. You have a list of all courses available to the students for Fall 2019. This includes the course title, description, number of credits, pre-requisits … etc. The URI would be[**https://api.mycollegesite.com/courses/2019/fall**](https://api.mycollegesite.com/courses/2019/fall)

At the same time, you want to access the courses for which a particular student has registered for fall 2019. Those would include the ID of the course, the grade, the grades for each homework, grades for each exam …

The URI would be [**https://api.mycollegesite.com/students/123456/courses/2019/fall**](https://api.mycollegesite.com/students/123456/courses/2019/fall)

Given that the resource returned by the first end-point is substantially different from the resource returned by the second end-point, should the resource name “**courses**” be used in both or is it better to have different resource names?  
Would be better to replace the first URI by  
[**https://api.mycollegesite.com/courses/curriculum/2019/fall**](https://api.mycollegesite.com/courses/curriculum/2019/fall) or  
[**https://api.mycollegesite.com/curriculum/courses/2019/fall**](https://api.mycollegesite.com/curriculum/courses/2019/fall)

[**https://api.mycollegesite.com/courses/2019/fall**](https://api.mycollegesite.com/courses/2019/fall)

[Reply](https://restfulapi.net/resource-naming/?replytocom=19408#respond)

* Sam says

[June 2, 2020 at 8:15 pm](https://restfulapi.net/resource-naming/#comment-19662)

First URI: **Courses seems to be the “collection” and year & semester are filters**.

* 1. /courses?**year=2019&semester=fall**

Second URI: Grades belong to each student so they could be a collection under students. **These are filtered by semeste**r. Grades could also be their own collection (maybe you want to get all grades for visualization?). If it’s useful for grades to exist independent from students, you can make grades their own collection.

* 1. /students/{id}/grades**?year=2019&semester=fall**
  2. /**grades?student\_id=123&year=2019&semester=fall**

I have a question about hierarchical relationship. How to define the number of layers, whether the parameter in the path are counted as one layer.  
For example, if “/device-management/managed-devices” has two hierarchies, then how many hierarchies in “/device-management/managed-devices/{id}”? three? or still two?

I am curious to know why you want to count the number of layers in the URI. But, anyway, for the given case, there are 3 layers, as the forward slashes are used to define relationships.  
“/device-management/managed-devices” is a collection  
while  
“/device-management/managed-devices/{id}” is **a document** (per definition of resource archetype)

1. **REST**

REST is acronym for **RE**presentational **S**tate **T**ransfer. It is **architectural style** for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous [dissertation](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm).

Like any other architectural style, REST also does have it’s own [**6** guiding constraints](https://restfulapi.net/rest-architectural-constraints/) which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

## Guiding Principles of REST

1. **Client–server**

* This constraint essentially means that client application and server application MUST be able to evolve separately without any dependency on each other. A client should know only resource URIs, and that’s all.

*Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.*

1. **Stateless** –

Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

* For example, **HTTP is a stateless protocol** layered on top of [TCP](https://en.wikipedia.org/wiki/Transmission_Control_Protocol), a stateful protocol, which is layered on top of [IP](https://en.wikipedia.org/wiki/Internet_Protocol), another stateless protocol, which is routed on a network that employs [BGP](https://en.wikipedia.org/wiki/BGP), another stateful protocol, to direct the IP packets riding on the network.
* This stacking of layers continues even above HTTP. As a workaround for the lack of a session layer in HTTP, [HTTP servers](https://en.wikipedia.org/wiki/HTTP_server) implement various [session management](https://en.wikipedia.org/wiki/Http_session#Web_server_session_management) methods,[[2]](https://en.wikipedia.org/wiki/Stateless_protocol#cite_note-2) typically utilizing a unique identifier in the [cookie](https://en.wikipedia.org/wiki/HTTP_cookie) or parameter that allows the server to track requests originating from the same client, and effectively creating a stateful protocol on top of HTTP.[[3]](https://en.wikipedia.org/wiki/Stateless_protocol#cite_note-statefulvstateless-3)
* Make all **client-server interactions stateless**. The server will not store anything about the latest HTTP request the client made. It will treat every request as new. No session, no history.
* If the client application needs to be a stateful application for the end-user, where user logs in once and do other authorized operations after that, then each request from the client should contain all the information necessary to service the request – including authentication and authorization details.
* **HTTP is a stateless protocol**, in other word the server will forget everything related to client/browser state. Although web applications have made it virtually look like stateful.
* **A stateless protocol can be forced to behave as if it were stateful**. This can be accomplished if the server sends the state to the client, and if the client to sends it back again to the server, every time.
* **There are three ways this may be accomplished in HTTP**:

1. One is **cookies**, in which case the state is sent and returned in HTTP headers.
2. The second is **URL extension**, in which case the state is sent as part of the URL as response.
3. The third is "hidden form fields", in which the state is sent to the client as part of the response, and returned to the server as part of a form's hidden data
4. **Cacheable** – Cache constraints require that **the data in a response** be implicitly or explicitly labeled **either as cacheable or non-cacheable**. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
5. **Uniform interface** –

* **A resource in the system should have only one logical URI,**
* **A resource should provide a way to fetch related or additional data**; A resource should contain **links (HATEOAS) pointing to relative URIs** to fetch related information.
* **the resource representations** across the system should follow **specific guidelines** such as naming conventions, link formats, or data format (XML or/and JSON).
* **All resources should be accessible through a common approach** such as HTTP GET and similarly modified using a consistent approach.

1. **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
2. **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

## Resource

The key abstraction of information in REST is a **resource**. Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources, a non-virtual object (e.g. a person), and so on. REST uses a **resource identifier** to identify the particular resource involved in an interaction between components.

**The state of the resource** at any particular **timestamp** is known as **resource representation**. A representation consists of **data**, **metadata** describing the data and **hypermedia** links which can help the clients in transition to the next desired state.

The **data format** of a representation is known as a [**media type**](https://www.iana.org/assignments/media-types/media-types.xhtml). The media type identifies a specification that defines how a representation is to be processed. **A truly RESTful API looks like hypertext**. Every addressable unit of information carries an address, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure).

Hypertext (or hypermedia) means the **simultaneous presentation of information and controls** such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions. Remember that hypertext does not need to be HTML (or XML or JSON) on a browser. Machines can follow links when they understand the data format and relationship types.

**Every media type defines a default processing model**. For example, **HTML** defines a rendering process for hypertext and the browser behavior around each element. It has no relation to the resource methods GET/PUT/POST/DELETE/… other than the fact that some media type elements will define a process model that goes like “anchor elements with an href attribute create a hypertext link that, when selected, invokes a retrieval request (GET) on the URI corresponding to the CDATA-encoded href attribute.”

## Resource Methods

Another important thing associated with REST is **resource methods** to be used to perform the desired transition. A large number of people wrongly relate resource methods to HTTP **GET/PUT/POST/DELETE** methods.

In simplest words, in the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs). The resources are acted upon by using a set of simple, well-defined operations. **The clients and servers exchange representations of resources** by using a standardized interface and protocol – typically HTTP.

**Resources are decoupled from their representation** so that their **content** can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. **Metadata** about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. And most importantly, every interaction with a resource is stateless.

1. **SCALABILITY AND HIGH AVAILABILITY**

One of the major reasons why HTTP scales so well is its Statelessness. Stateless protocol eases the replication concerns, as the state itself doesn't need to be stored on the server.

Stateful protocols are logically heavy to implement in Internet reliably. Stateless servers are also easily scalable, while for stateful servers scalability is problematic. Stateless request can be sent to any node, at any time, while with Stateful this is not a case.

HTTP as Stateless protocol increases availability for stateless web applications, which otherwise would be difficult or impossible to implement. **If there is connection lost, there is no state that is lost, simple request resend will resolve the problem. Stateless requests are also cacheable**.

5

HTTP is a stateless protocol. All the web-based applications are also stateless.

When a request is send to the server, a connection is established between client and server. The server receives the request, processes the request and sends back the response and then, the connection will be closed.

If another request will be sent, after that, it will be treated as a new request and a new connection will be established.

In order to make HTTP stateful, we use ***session management*** *techniques*. So that, it uses the data coming from previous request while processing present request i.e, **it uses the same connection** for a series of client server interactions.

The session management techniques are:

1. hidden form field
2. cookie
3. session
4. URL-rewriting

*No client context shall be stored on the server between requests. The client is responsible for managing the state of the application.*

#### **Cacheable**

In today’s world, the caching of data and responses is of utmost importance wherever they are applicable/possible. The webpage you are reading here is also a cached version of the HTML page. Caching brings performance improvement for the client-side and **better scope for scalability** for a server because the load has reduced.

In REST, **caching shall be applied to resources when applicable,** and then **these resources MUST declare themselves cacheable**. Caching can be implemented on the server or client-side.

*Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.*

#### **Layered system**

REST allows you to use a layered system architecture where you deploy the APIs on server A, and **store data on server B and authenticate requests in Server C**, for example. A client cannot ordinarily tell whether it is connected directly to the end server or an intermediary along the way.

#### Code on demand (optional)

Well, this constraint is optional. Most of the time, you will be sending the static representations of resources in the form of XML or JSON. But when you need to, you are free to return executable code to support a part of your application, e.g., clients may call your API to get a UI widget rendering code. It is permitted.

*All the above constraints help you build a truly RESTful API, and you should follow them. Still, at times, you may find yourself violating one or two constraints. Do not worry; you are still making a RESTful API – but not “truly RESTful.”*

Notice that all the above constraints are most closely related to WWW (the web). Using RESTful APIs, you can do the same thing with your web services what you do to web pages.