Rest API:

Idempotent REST APIs

In the context of REST APIs, when making multiple identical requests has the 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. They can write client code in such a way that there can be duplicate requests as well. 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. It 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 idempotent operation.

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` is NOT idempotent.
2. `GET`, `PUT`, `DELETE`, `HEAD`, `OPTIONS` and `TRACE` are idempotent.

Let's analyze how above HTTP methods end up being idempotent – any 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 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:

```
DELETE /item/last
```

In the above case, calling operation N times will delete N resources – hence `DELETE` is not idempotent in this case. In this case, a good suggestion might be to change above API to POST – because POST is not idempotent.

```
POST /item/last
```

Now, this is closer to HTTP spec – hence more REST compliant.

References:

[Rfc 2616]
[SO Thread]

Why we need post if put did create and update both?

So if you only want ONE resource to be affected regardless of how many times you make a call, then PUT is the right command. If you want 10 or 20 resources to be affected (created), then you'd call the POST command 10 or 20 times, respectively.

PUT vs. POST in REST

`POST` is used to **create**.

`PUT` is used to **create or update**.

So, which one should be used to create a resource? Or one needs to support both?

You can find assertions on the web that say

- **POST** should be used to create a resource, and **PUT** should be used to modify one
- **PUT** should be used to create a resource, and **POST** should be used to modify one

Neither is quite right.

Better is to choose between PUT and POST based on idempotence of the action.

PUT implies putting a resource - completely replacing whatever is available at the given URL with a different thing. By definition, a PUT is idempotent. Do it as many times as you like, and the result is the same. `x=5` is idempotent. You can PUT a resource whether it previously exists, or not (eg, to Create, or to Update)!

POST updates a resource, adds a subsidiary resource, or causes a change. A POST is not idempotent, in the way that `x++` is not idempotent.

By this argument, PUT is for creating when you know the URL of the thing you will create. POST can be used to create when you know the URL of the "factory" or manager for the category of things you want to create.

so:

```
POST /expense-report
```

or:

```
PUT  /expense-report/{uuid}
```

Note:

Put request url should have a unique id of resource and if resource is present it will be updated or if not present a new resource is created.(new resource creation can be blocked by giving 404 code saying resource not found, if create is being supported by POST request)

In case of create using PUT Client need to have headache of using unique uuid eachtime to create.

But if we use POST (where we do not need to provide uuid in request url, uuid is generated on server side) we don't need to have such headeache.

Response code:

REST API - Response Codes and Statuses

Code	Status	Description
200	OK	The request was successfully completed.
201	Created	A new resource was successfully created.
204	No Content	The server has successfully fulfilled the request and that there is no additional content to send in the response payload body.
400	Bad Request	The request was invalid.
401	Unauthorized	The request did not include an authentication token or the authentication token was expired.
403	Forbidden	The client did not have permission to access the requested resource.
404	Not Found	The requested resource was not found.
409	Conflict	The request could not be completed due to a conflict. For example, POSTContentStore Folder API cannot complete if the given file or folder name already exists in the parent location.
500	Internal Server Error	A generic error message, given when an unexpected condition was encountered and no more specific message is suitable
502	Bad Gateway	It means that one server on the internet received an invalid **response** from another server. Or The server was acting as a gateway or proxy and received an invalid response from the upstream server

406 (Not Acceptable)

The 406 error response indicates that the API is not able to generate any of the client's preferred media types, as indicated by the Accept request header. For example, a client request for data formatted as `application/xml` will receive a 406 response if the API is only willing to format data as `application/json`.

415 (Unsupported Media Type)

The 415 error response indicates that the API 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`.

Head vs Option:

- `OPTIONS` Used to retrieve the available HTTP verbs for a given resource?
- `HEAD` Used to determine whether a given resource is available?

`OPTIONS` method returns info about **API** (methods/content type)
`HEAD` method returns info about **resource** (version/length/type)
Server response

OPTIONS

```
HTTP/1.1200 OK
Allow:GET,HEAD,POST,OPTIONS,TRACE
Content-Type: text/html; charset=UTF-8
Date:Wed,08May201310:24:43 GMT
Content-Length:0
```

HEAD

```
HTTP/1.1200 OK
Accept-Ranges: bytes
Content-Type: text/html; charset=UTF-8
Date:Wed,08May201310:12:29 GMT
ETag:"780602-4f6-4db31b2978ec0"
```

```
Last-Modified:Thu,25Apr201316:13:23 GMT
Content-Length:1270
```

What is a Resource in REST?

REST architecture treats every content as a resource. These resources can be text files, html pages, images, videos or dynamic business data. REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs.

How to represent a resource in REST?

REST uses various representations to represent a resource where text, JSON, XML. XML and JSON are the most popular representations of resources.

What are the core components of a HTTP Request?

A HTTP Request has five major parts −

- **Verb** − Indicate HTTP methods such as GET, POST, DELETE, PUT etc.
- **URI** − Uniform Resource Identifier (URI) to identify the resource on server.
- **HTTP Version** − Indicate HTTP version, for example HTTP v1.1 .
- **Request Header** − Contains metadata for the HTTP Request message as key-value pairs. For example, client (or browser) type, format supported by client, format of message body, cache settings etc.
- **Request Body** − Message content or Resource representation.

What are the core components of a HTTP response?

A HTTP Response has four major parts −

- **Status/Response Code** − Indicate Server status for the requested resource. For example 404 means resource not found and 200 means response is ok.
- **HTTP Version** − Indicate HTTP version, for example HTTP v1.1 .
- **Response Header** − Contains metadata for the HTTP Response message as key-value pairs. For example, content length, content type, response date, server type etc.
- **Response Body** − Response message content or Resource representation.

What is statelessness in RESTful Webservices?

As per REST architecture, a RESTful web service should not keep a client state on server. This restriction is called statelessness. It is responsibility of the client to pass its context to server and

then server can store this context to process client's further request. For example, session maintained by server is identified by session identifier passed by the client.

What are the advantages of statelessness in RESTful Webservices?

Following are the benefits of statelessness in RESTful web services −

- Web services can treat each method request independently.
- Web services need not to maintain client's previous interactions. It simplifies application design.
- As HTTP is itself a statelessness protocol, RESTful Web services work seamlessly with HTTP protocol.

What are the disadvantages of statelessness in RESTful Webservices?

Following is the disadvantage of statelessness in RESTful web services −

Web services need to get extra information in each request and then interpret to get the client's state in case client interactions are to be taken care of.

What do you mean by idempotent operation?

Idempotent operations means their result will always same no matter how many times these operations are invoked.

Which type of Webservices methods are to be idempotent?

PUT and DELETE operations are idempotent.

Which type of Webservices methods are to be read only?

GET operations are read only and are safe.

What is the difference between PUT and POST operations?

PUT and POST operation are nearly same with the difference lying only in the result where PUT operation is idempotent and POST operation can cause different result.

What is caching?

Caching refers to storing server response in client itself so that a client needs not to make server request for same resource again and again. A server response should have information about

how a caching is to be done so that a client caches response for a period of time or never caches the server response.

Which header of HTTP response provides control over caching?

Cache-Control is the primary header to control caching.

Which header of HTTP response sets expiration date and time of caching?

Expires header sets expiration date and time of caching.

Which directive of Cache Control Header of HTTP response indicates that resource is not cacheable?

no-cache/no-store directive indicates that resource is not cacheable.

Caching REST API Response

Caching 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 all the way to the service (or origin server as it is formally known).

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 following ways:

- Reduce bandwidth
- Reduce latency

- Reduce load on servers
- Hide network failures

Caching in REST APIs

Being cacheable is one of architectural constraints of REST.
GET requests should be cacheable 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. 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
```

Which header of HTTP response, provides the date and time of the resource when it was created?

Date header provides the date and time of the resource when it was created.

Which header of HTTP response, provides the date and time of the resource when it was last modified?

Last Modified header provides the date and time of the resource when it was last modified.