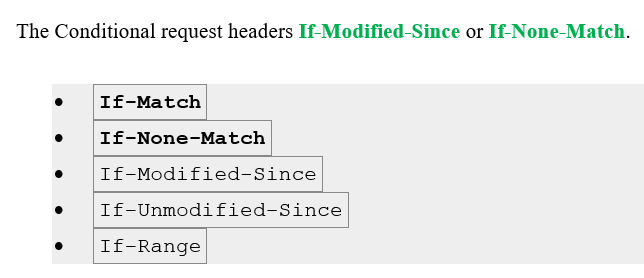
4/12/22

1. Status code
2. REST
3. **Idempotent REST APIs**
4. **Caching REST API Response**
5. HTTP method
6. headers

#### By default, The following responses are cacheable.

#### 200,204, [**301 (Moved Permanently)**](https://restfulapi.net/http-status-301-moved-permanently/)**,308**

* No payload body: 204, 304,
* Avoid lost updates problem (last updater wins), by including Etag.
* 

**Status code**

***200 ok***

**201 Created**

**202 Accepted**

**204 No Content**

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

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

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

**308 Permanent Redirect**

* Can not change POST to GET; while 301 can change.

**4xx Client Error**

**400 (Bad Request)**

***401 (Unauthorized)***

***403 (Forbidden)***

***404 (Not Found)***

***405 (Method Not Allowed)***

***406 (Not Acceptable)***

**Accept: text/html**

**Accept-Encoding: gzip, compress**

***412 Precondition Failed***

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

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

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

# ***502 Bad Gateway***

***503 (Service Unavailable Error)***

**REST** is acronym for **RE**presentational **S**tate **T**ransfer. It is **architectural style** for **distributed hypermedia systems**

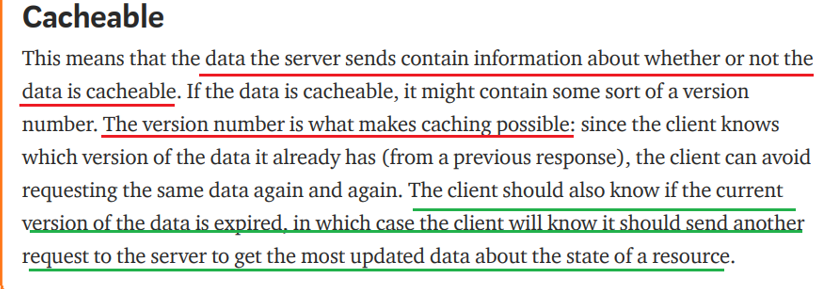
* **Client–server**
* **Stateless**
* **Cacheable**
* **Uniform interface**
* **Layered system**
* **Code on demand (optional)**
* **The state of the resource** at any particular **timestamp** is known as **resource representation**. A representation consists of **data**, **metadata** describing the 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).
* 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.

**Idempotent REST APIs**

* **when making multiple identical requests** has t**he same effect as making a single request** – then that REST API is called **idempotent**.
* **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.*
* benefits of idempotence is **reliable caching** of responses on the network and **reliable updates of resources from retries** when the original request fails
* **POST, Patch is NOT idempotent.**
* **GET, PUT, DELETE, HEAD, OPTIONS and TRACE are idempotent.**

**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).



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.

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.

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**

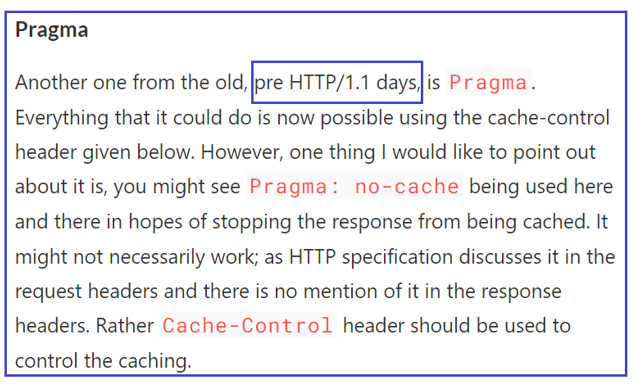
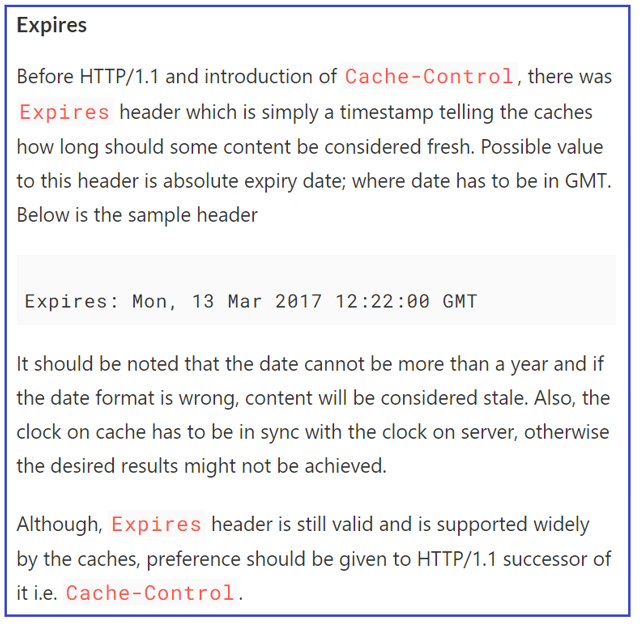
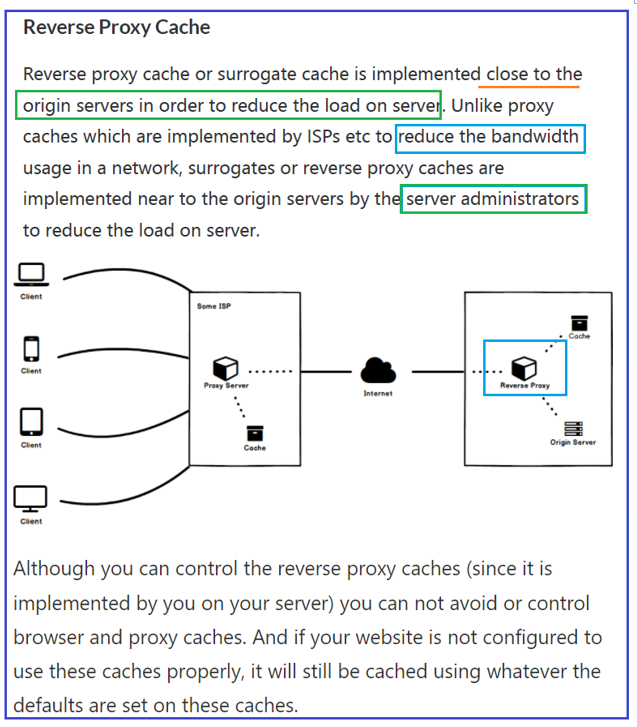
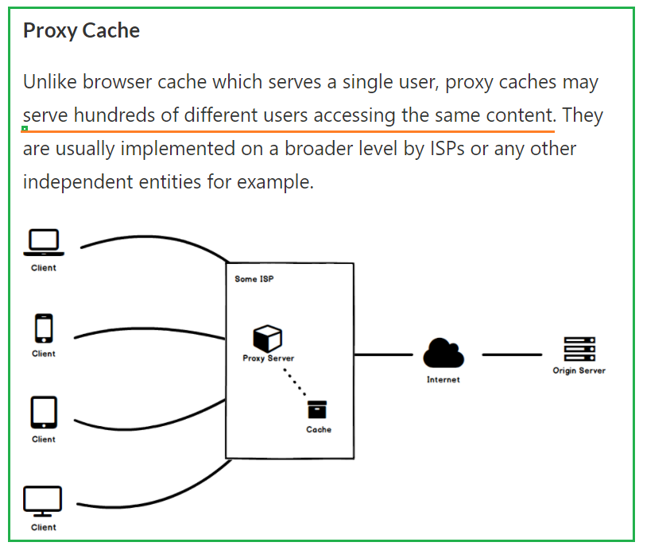
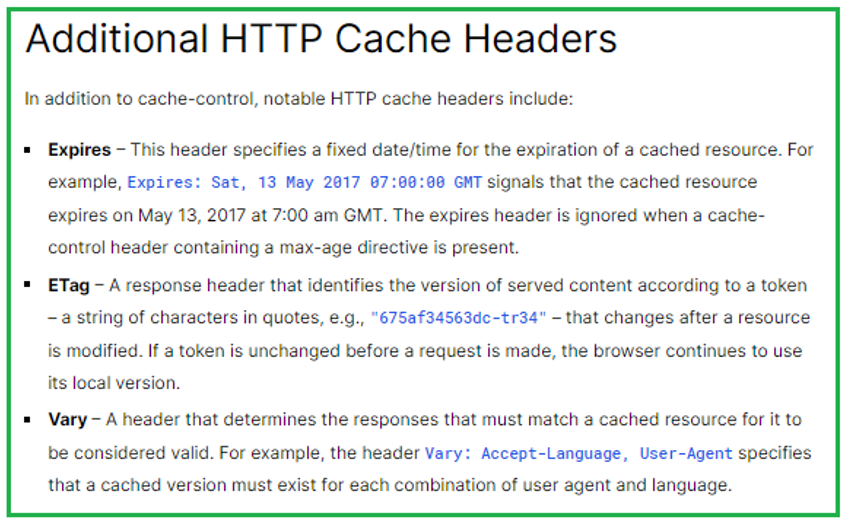
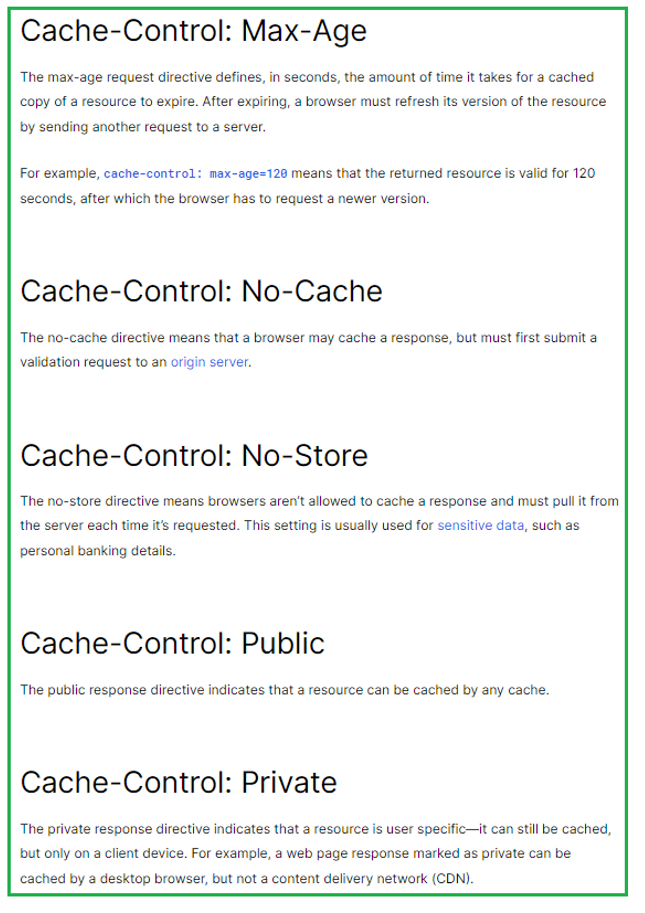
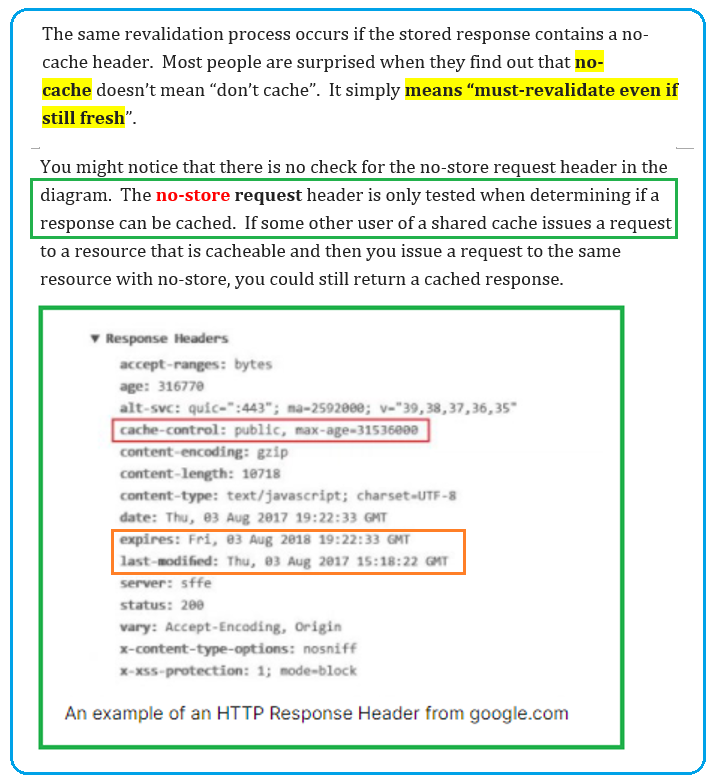
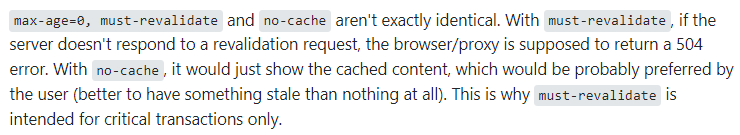
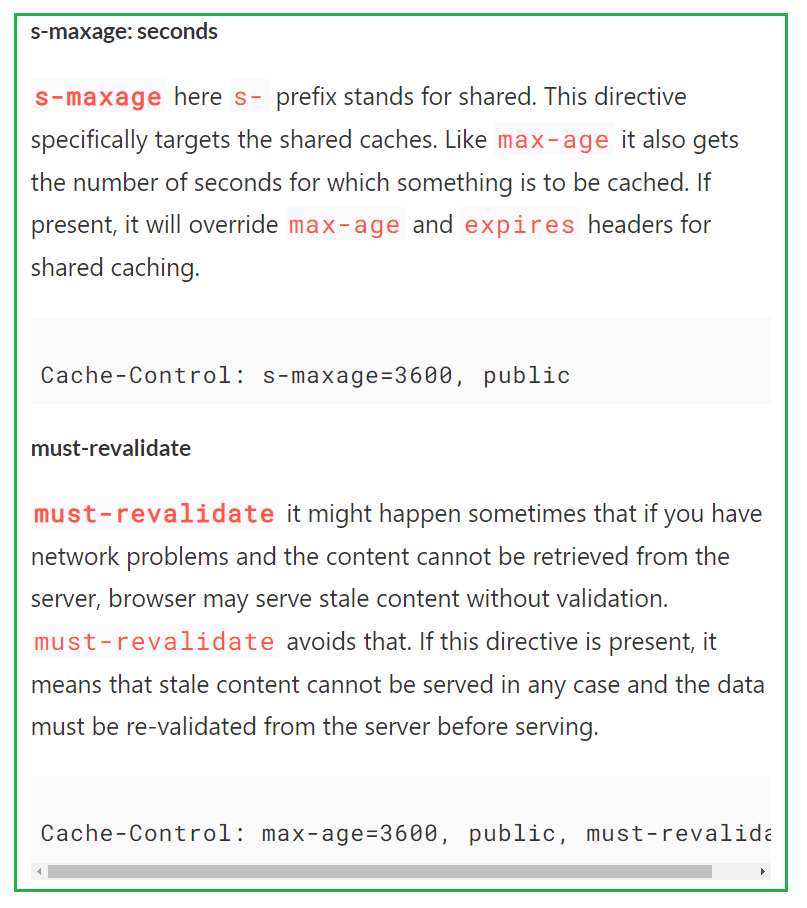
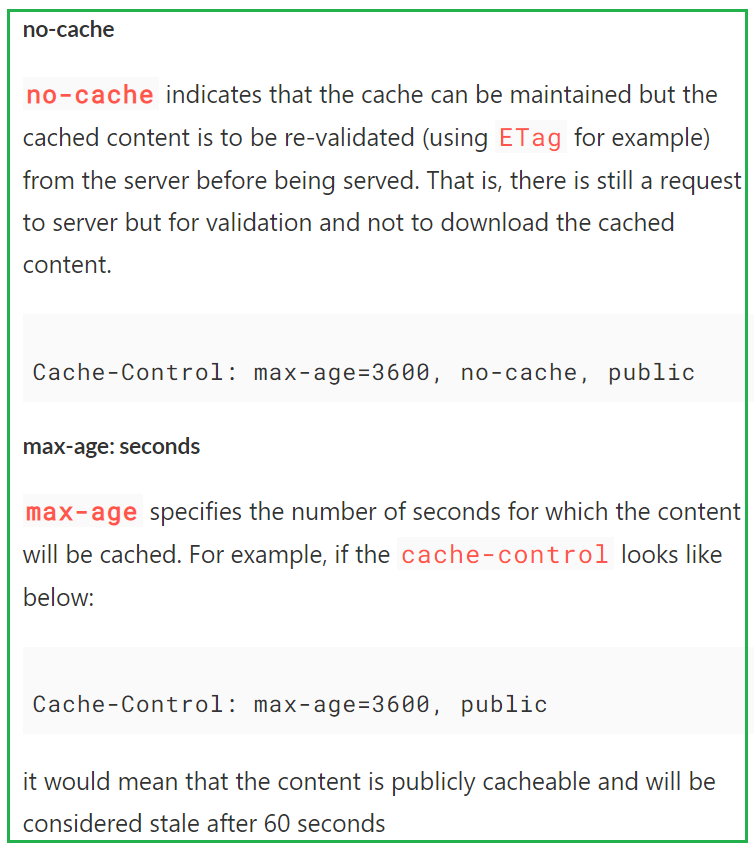
* The header value comprises one or more comma-separated [directives](https://tools.ietf.org/html/rfc7234#page-24). These directives determine whether a response is cacheable, and if so, by whom, and for how long e.g. max-age or s-maxage directives.
* **Cache**-**Control**: **max-age**=3600
* Cacheable 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**

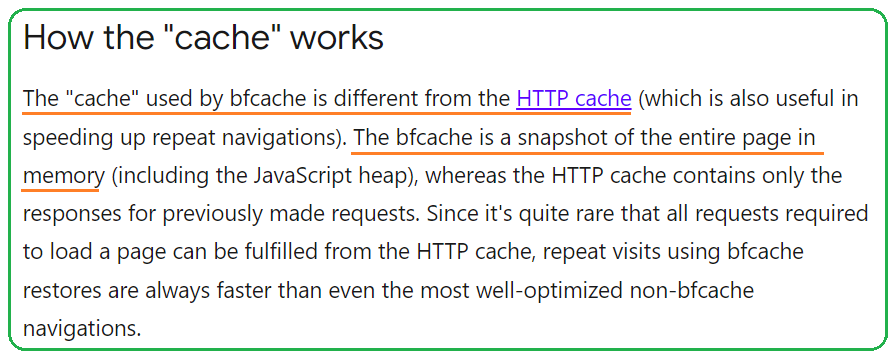
* Whereas a response’s Date header indicates when the response was generated, the Last-Modified header indicates when the associated resource last changed. The Last-Modified value cannot be less than the Date value.
* **Last-Modified**: Fri, 10 May 2016 09:17:49 IST

****

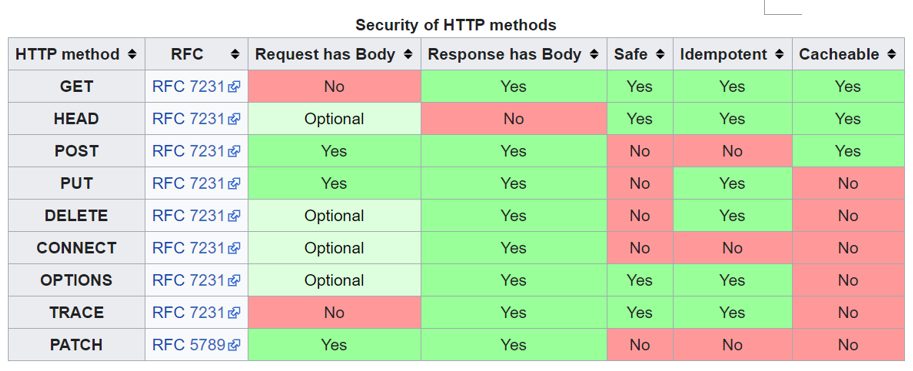
# **Back/forward cache**

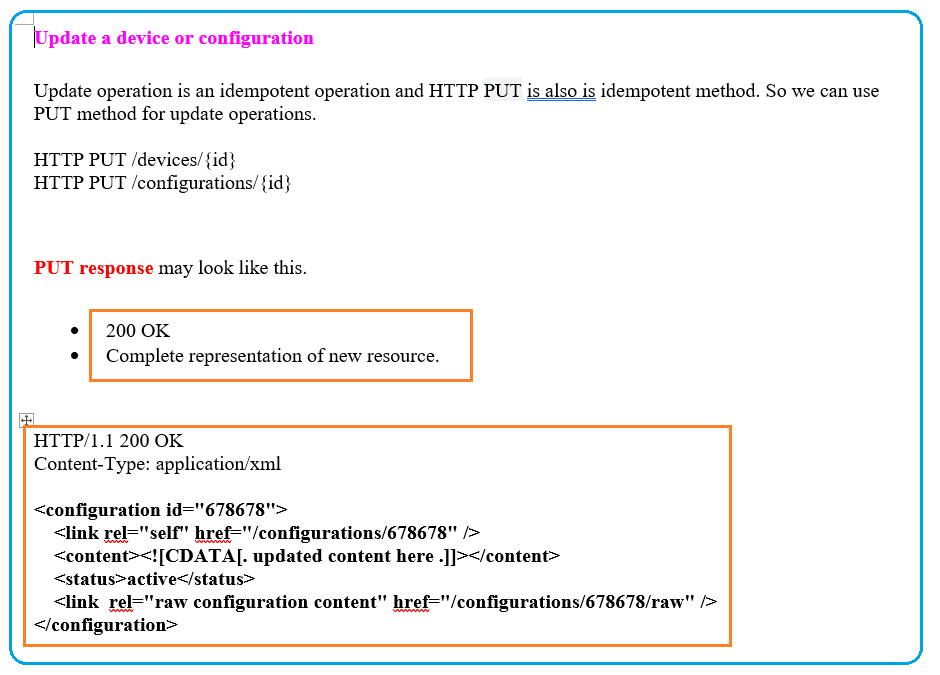
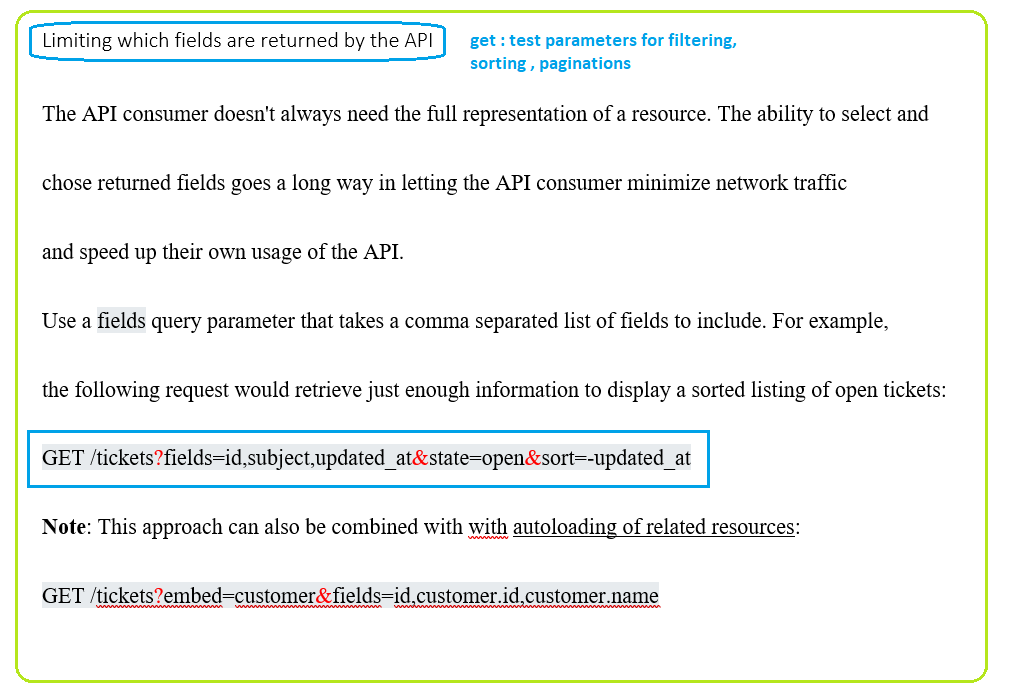
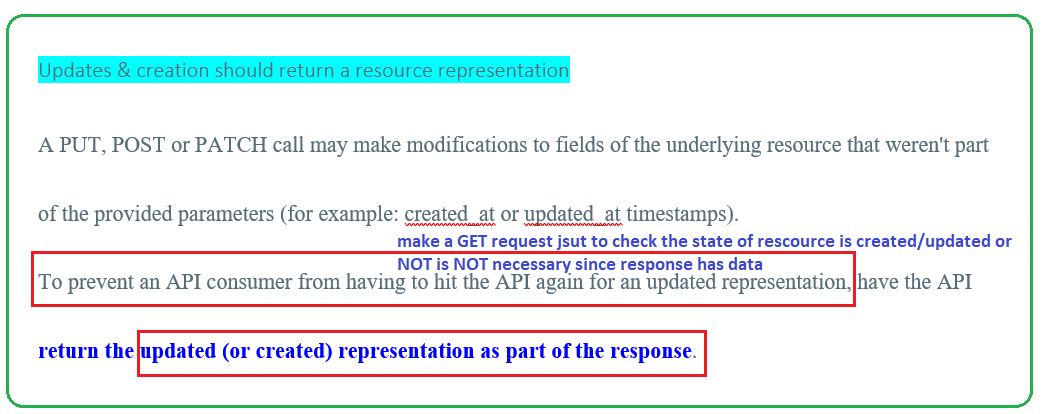
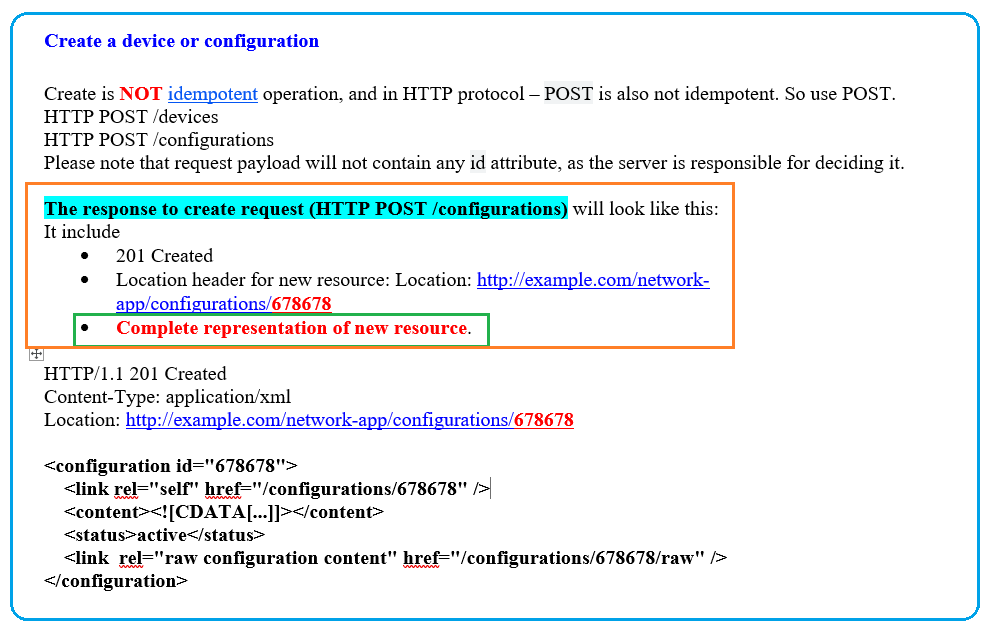
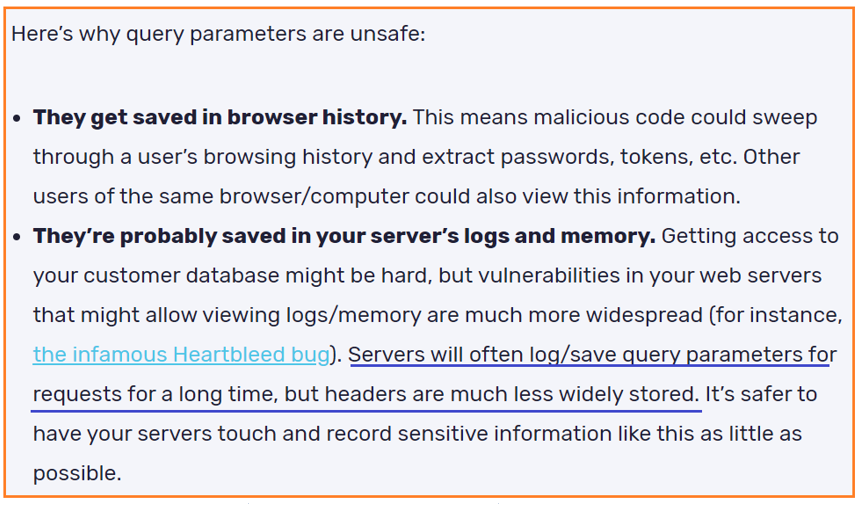
Optimize your pages for instant loads when using the browser's back and forward buttons.

Back/forward cache (or bfcache) is a **browser optimization** that enables instant back and forward navigation. It significantly improves the browsing experience for users—especially those with slower networks or devices.

****

**HTTP method**

****

** *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****.* **

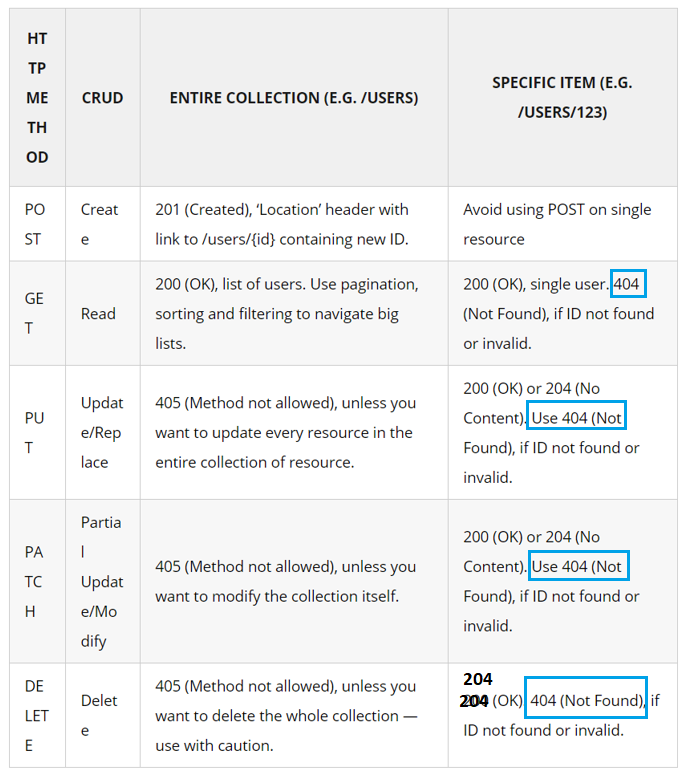
## **Lost update problem**

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.

**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**.

****

**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.

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

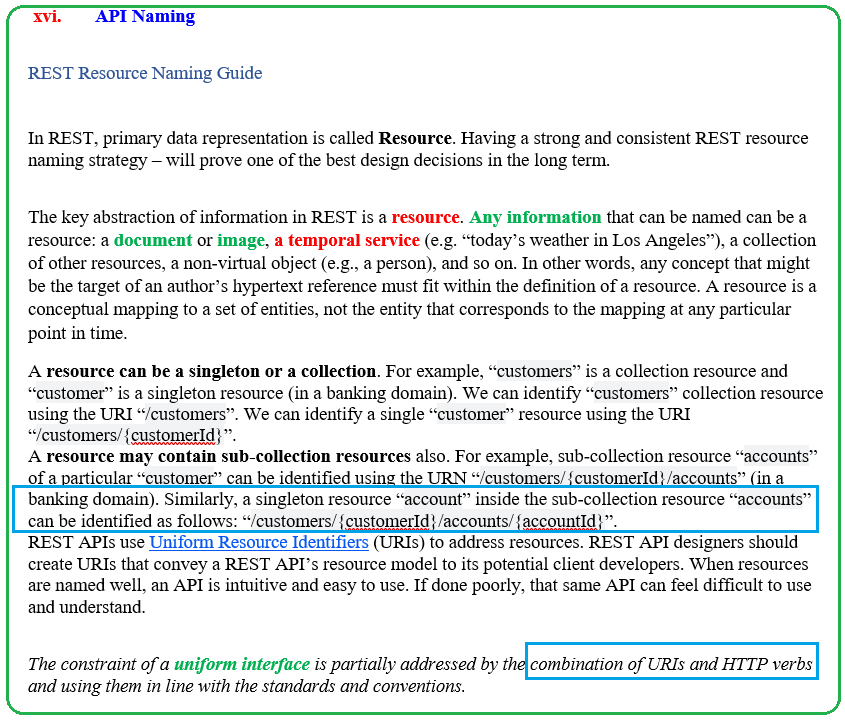
Check for Unexpected Maintenance

Server Connectivity Issues

Improper Firewall Configuration

Check the Logs – Nearly every web application will keep some form of server-side logs.

Application Code or Script Bugs

****

**Headers**

**Content-Type:** application/json, client-side media type of resource sent to server

“text/plain”, “text/html”,

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

“image/gif”, and “image/jpeg”.

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.

**Accept:** application/json

**406 (Not Acceptable)** if server does Not support the media type requested

### **Client side request 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**

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.

## **Location Header**

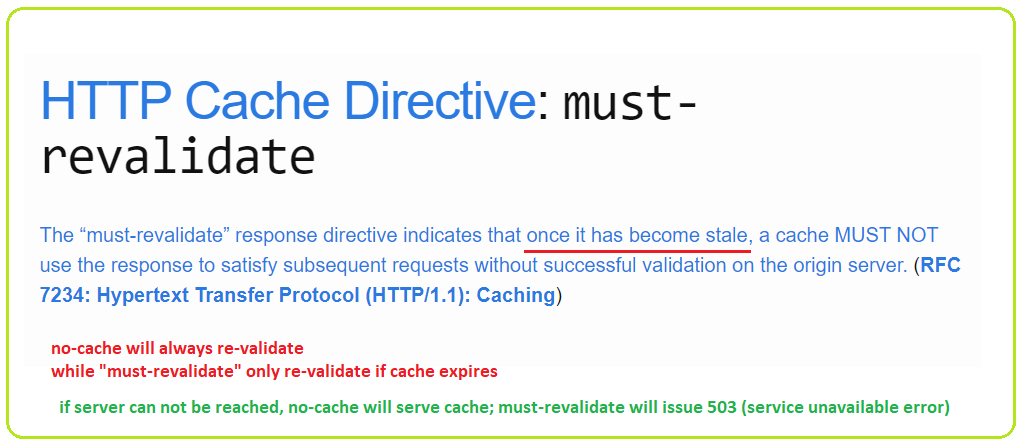
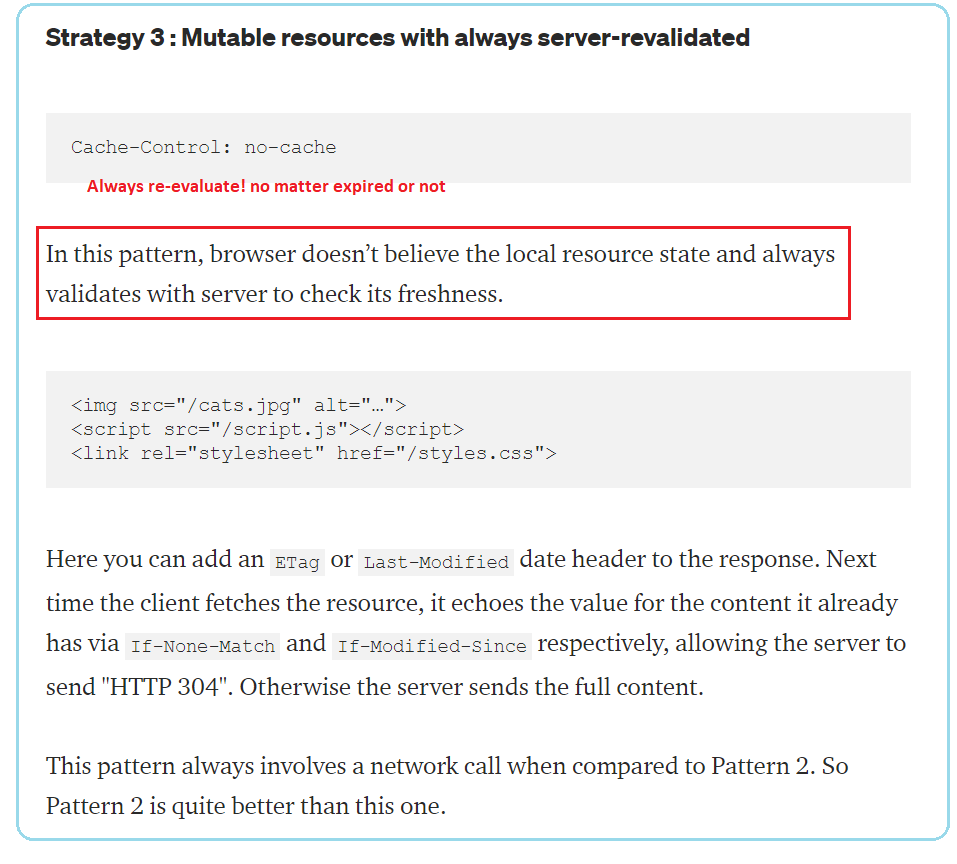
The server SHOULD generate a Location header field in the response containing the new location of the resource.

#### **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** |

******