FREE eBook

LEARNING HTTP

Free unaffiliated eBook created from
Stack Overflow contributors.

#http

Table of Contents

About 1

Chapter 1: Getting started with HTTP 2

Remarks 2

Versions 2

Examples 2

HTTP requests and responses 2

HTTP/1.0 3

HTTP/1.1 3

HTTP/2 4

HTTP/0.9 4

Chapter 2: Authentication 6

Parameters 6

Remarks 6

Examples 6

HTTP Basic Authentication 6

Chapter 3: Caching HTTP responses 8

Remarks 8

Glossary 8

Examples 8

Cache response for everyone for 1 year 8

Cache personalized response for 1 minute 8

Stop use of cached resources without checking with the server first 9

Request responses not to be stored at all 9

Obsolete, redundant and non-standard headers 9

Changing cached resources 9

Chapter 4: Cross Origin and Access Control 11

Remarks 11

Examples 11

Client: sending a cross-origin resource sharing (CORS) request 11

Server: responding to a CORS request 11

Permitting user credentials or session ... 11
Preflighting requests ... 12
Server: responding to preflight requests ... 12
Chapter 5: HTTP for APIs ... 14
Remarks ... 14
Examples ... 14
Create a resource ... 14
Edit a resource ... 15
Full updates ... 15
Side-Effects ... 17
Partial updates ... 17
Partial update with overlapping state ... 17
Patching partial data ... 19
Error Handling ... 20
Delete a resource ... 21
List resources ... 21
Chapter 6: HTTP requests ... 24
Parameters ... 24
Remarks ... 24
Examples ... 24
Sending a minimal HTTP request manually using Telnet ... 24
Basic request format ... 26
Request header fields ... 26
Message bodies ... 27
Chapter 7: HTTP responses ... 28
Parameters ... 28
Examples ... 30
Basic response format ... 30
Additional Headers ... 31
Message Bodies ... 32
Chapter 8: HTTP Status Codes ... 33
Introduction ... 33

Remarks 33
Examples 33
500 Internal Server Error 33
404 Not Found 33
Denying access to protected files 34
Successful request 34
Responding to a conditional request for cached content 34
Top 10 HTTP Status Code 34
2xx Success 34
3xx Redirection 35
4xx Client Error 35
5xx Server Error 35
Chapter 9: Response encodings and compression 36
Examples 36
HTTP compression 36
Multiple compression methods 36
gzip compression 36
Credits 38

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: http

It is an unofficial and free HTTP ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official HTTP.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with HTTP

Remarks

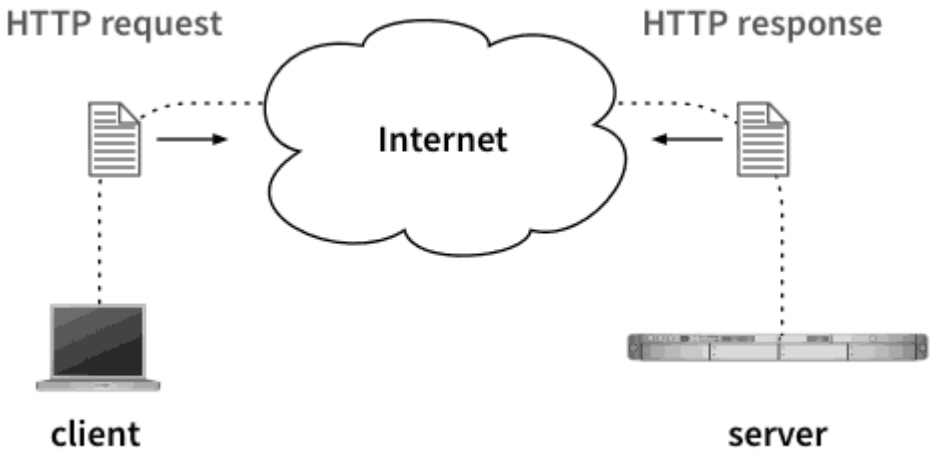
Hypertext Transfer Protocol (HTTP) uses a client-request/server-response model. HTTP is a stateless protocol, which means it does not require the server to retain information or status about each user for the duration of multiple requests. However, for performance reasons and to avoid TCP' connection-latency issues, techniques like Persistent, Parallel or Pipelined connections may be used.

Versions

Version	Note(s)	Estimated Release Date
HTTP/0.9	"As Implemented"	1991-01-01
HTTP/1.0	First version of HTTP/1.0, last version that has not made it into an RFC	1992-01-01
HTTP/1.0 r1	First official RFC for HTTP	1996-05-01
HTTP/1.1	Improvements in connection handling, support for name-based virtual hosts	1997-01-01
HTTP/1.1 r1	Disambiguous keyword usage cleaned up, possible issues with message framing fixed	1999-06-01
HTTP/1.1 r2	Major overhaul	2014-06-01
HTTP/2	First spec for HTTP/2	2015-05-01

Examples

HTTP requests and responses



HTTP describes how an HTTP client, such as a web browser, sends an HTTP request via a network to an HTTP server, which then sends an HTTP response back to the client.

The HTTP request is typically either a request for an online resource, such as a web page or image, but may also include additional information, such as data entered on a form. The HTTP response is typically a representation of an online resource, such as a web page or image.

HTTP/1.0

HTTP/1.0 was described in RFC 1945.

HTTP/1.0 does not have some features that are today de-facto required on the Web, such as the `Host` header for virtual hosts.

However, HTTP clients and servers sometimes still declare they use HTTP/1.0 if they have incomplete implementation of the HTTP/1.1 protocol (e.g. without chunked transfer encoding or pipelining), or compatibility is considered more important than performance (e.g. when connecting to local proxy servers).

```
GET / HTTP/1.0
User-Agent: example/1

HTTP/1.0 200 OK
Content-Type: text/plain

Hello
```

HTTP/1.1

HTTP/1.1 has originally been specified in 1999 in RFC 2616 (protocol) and RFC 2617 (authentication), but these documents are now obsolete and should not be used as a reference:

> Don’t use RFC2616. Delete it from your hard drives, bookmarks, and burn (or responsibly recycle) any copies that are printed out.
>
> — Mark Nottingham, chair of the HTTP WG

The up-to-date specification of HTTP/1.1, that matches how HTTP is implemented today, is in new RFCs 723x:

- RFC 7230: Message Syntax and Routing
- RFC 7231: Semantics and Content
- RFC 7232: Conditional Requests
- RFC 7233: Range Requests
- RFC 7234: Caching
- RFC 7235: Authentication

HTTP/1.1 added, among other features:

- chunked transfer encoding, which allows servers to reliably send responses of unknown size,
- persistent TCP/IP connections (which were non-standard extension in HTTP/1.0),
- range requests used for resuming downloads,
- cache control.

HTTP/1.1 tried to introduce pipelining, which allowed HTTP clients to reduce request-response latency by sending multiple requests at once without waiting for responses. Unfortunately, this feature was never correctly implemented in some proxies, causing pipelined connections to drop or reorder responses.

```
GET / HTTP/1.0
User-Agent: example/1
Host: example.com

HTTP/1.0 200 OK
Content-Type: text/plain
Content-Length: 6
Connection: close

Hello
```

HTTP/2

HTTP/2 (RFC 7540) changed on-the-wire format of HTTP from a simple text-based request and response headers to binary data format sent in frames. HTTP/2 supports compression of the headers (HPACK).

This reduced overhead of requests, and enabled receiving of multiple responses simultaneously over a single TCP/IP connection.

Despite big changes in the data format, HTTP/2 still uses HTTP headers, and requests and responses can be accurately translated between HTTP/1.1 and 2.

HTTP/0.9

The first version of HTTP that came into existence is 0.9, often referred to as "HTTP As Implemented." A common description of 0.9 is "a subsect of the full HTTP [i.e. 1.0] protocol."

However, this greatly fails to illustrate the disparity in capabilities between 0.9 and 1.0.

Neither requests nor responses in 0.9 feature headers. Requests consist of a single CRLF-terminated line of `GET`, followed by a space, followed by the requested resource URL. Responses are expected to be a single HTML document. The end of said document is marked by dropping the connection server-side. There are no facilities to indicate success or failure of an operation. The only interactive property is the search string which is closely tied to the `<isindex>` HTML tag.

Usage of HTTP/0.9 is nowadays exceptionally rare. It is occasionally seen on embedded systems as an alternative to tftp.

Read Getting started with HTTP online: https://riptutorial.com/http/topic/984/getting-started-with-http

Chapter 2: Authentication

Parameters

Parameter	Details
Response status	`401` if the origin server requires authentication, `407` if an intermediate proxy requires authentication
Response headers	`WWW-Authenticate` by the origin server, `Proxy-Authenticate` by an intermediate proxy
Request headers	`Authorization` for authorization against an origin server, `Proxy-Authorization` against an intermediate proxy
Authentication scheme	`Basic` for Basic Authentication, but others such as `Digest` and `SPNEGO` can be used. See the HTTP Authentication Schemes Registry.
Realm	A name of the protected space on the server; a server can have multiple such spaces, each with a distinct name and authentication mechanisms.
Credentials	For `Basic`: username and password separated by a colon, base64-encoded; for example, `username:password` base64-encoded is `dXNlcm5hbWU6cGFzc3dvcmQ=`

Remarks

Basic Authentication is defined in RFC2617. It can be used to authenticate against the origin server after receiving a `401 Unauthorized` as well as against a proxy server after a `407 (Proxy Authentication Required)`. In the (decoded) credentials, the password starts after the first colon. Therefore the username cannot contain a colon, but the password can.

Examples

HTTP Basic Authentication

HTTP Basic Authentication provides a straightforward mechanism for authentication. Credentials are sent in plain text, and so is insecure by default. Successful authentication proceeds as follows.

The client requests a page for which access is restricted:

```
GET /secret
```

The server responds with status code `401 Unauthorized` and requests the client to authenticate:

```
401 Unauthorized
WWW-Authenticate: Basic realm="Secret Page"
```

The client sends the `Authorization` header. The credentials are `username:password` base64 encoded:

```
GET /secret
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

The server accepts the credentials and responds with the page content:

```
HTTP/1.1 200 OK
```

Read Authentication online: https://riptutorial.com/http/topic/3286/authentication

Chapter 3: Caching HTTP responses

Remarks

Responses are cached separately for each URL and each HTTP method.

HTTP caching is defined in RFC 7234.

Glossary

- **fresh** — state of a cached response, which hasn't expired yet. Typically, a fresh response can *satisfy* requests without a need to contact the server the response originated from.
- **stale** — state of a cached response, which is past its expiration date. Typically, stale responses can't be used to *satisfy* a request without checking with the server whether it's still valid.
- **satisfy** — cached response *satisfies* a request when all conditions in the request match the cached response, e.g. they have the same HTTP method and URL, the response is fresh or the request allows stale responses, request headers match headers listed in response's `Vary` header, etc.
- **revalidation** — checking whether a cached response is fresh. This is usually done with a *conditional request* containing `If-Modified-Since` or `If-None-Match` and response status `304`.
- **private cache** — cache for a single user, e.g. in a web browser. Private caches can store personalized responses.
- **public cache** — cache shared between many users, e.g. in a proxy server. Such cache can send the same response to multiple users.

Examples

Cache response for everyone for 1 year

```
Cache-Control: public, max-age=31536000
```

`public` means the response is the same for all users (it does not contain any personalized information). `max-age` is in seconds from now. 31536000 = 60 * 60 * 24 * 365.

This is recommended for static assets that are never meant to change.

Cache personalized response for 1 minute

```
Cache-Control: private, max-age=60
```

`private` specifies that the response can be cached only for user who requested the resource, and can't be reused when other users request the same resource. This is appropriate for responses that depend on cookies.

Stop use of cached resources without checking with the server first

```
Cache-Control: no-cache
```

The client will behave as if the response was not cached. This is appropriate for resources that can unpredictably change at any time, and which users must always see in the latest version.

Responses with `no-cache` will be slower (high latency) due to need to contact the server every time they're used.

However, to save bandwidth, the clients *may* still store such responses. Responses with `no-cache` won't be used to satisfy requests without contacting the server each time to check whether the cached response can be reused.

Request responses not to be stored at all

```
 Cache-control: no-store
```

Instructs clients no to cache the response in any way, and to forget it at soon as possible.

This directive was originally designed for sensitive data (today HTTPS should be used instead), but can be used to avoid polluting caches with responses that can't be reused.

It's appropriate only in specific cases where the response data is always different, e.g. an API endpoint that returns a large random number. Otherwise, `no-cache` and revalidation can be used to have a behavior of "uncacheable" response, while still being able to save some bandwidth.

Obsolete, redundant and non-standard headers

- `Expires` — specifies date when the resource becomes stale. It relies on servers and clients having accurate clocks and supporting time zones correctly. `Cache-control: max-age` takes precedence over `Expires`, and is generally more reliable.
- `post-check` and `pre-check` directives are non-standard Internet Explorer extensions that enable use of stale responses. The standard alternative is `stale-while-revalidate`.
- `Pragma: no-cache` — obsoleted in 1999. `Cache-control` should be used instead.

Changing cached resources

The easiest method to bypass cache is to change the URL. This is used as a best practice when the URL contains a version or a checksum of the resource, e.g.

```
http://example.com/image.png?version=1
http://example.com/image.png?version=2
```

These two URLs will be cached separately, so even if `…?version=1` was cached *forever*, a new copy could be immediately retrieved as `…?version=2`.

Please don't use random URLs to bypass caches. Use `Cache-control: no-cache` or `Cache-control: no-store` instead. If responses with random URLs are sent without the `no-store` directive, they will be unnecessarily stored in caches and push out more useful responses out of the cache, degrading performance of the entire cache.

Read Caching HTTP responses online: https://riptutorial.com/http/topic/3296/caching-http-responses

Chapter 4: Cross Origin and Access Control

Remarks

Cross-origin resource sharing is designed to allow dynamic requests between domains, often using techniques such as AJAX. While the scripting does most of the work, the HTTP server must support the request using the correct headers.

Examples

Client: sending a cross-origin resource sharing (CORS) request

A cross-origin *request* must be sent including the `Origin` header. This indicates from where the request originated. For example, a cross-origin request from `http://example.com` to `http://example.org` would look like this:

```
GET /cors HTTP/1.1
Host: example.org
Origin: example.com
```

The server will use this value to determine if the request is authorized.

Server: responding to a CORS request

The response to a CORS request must include an `Access-Control-Allow-Origin` header, which dictates what origins are allowed to use the CORS resource. This header can take one of three values:

- An origin. Doing this permits requests from *that origin only*.
- The character `*`. This permits requests from *any origin*.
- The string `null`. This permits *no CORS requests*.

For example, on reception of a CORS request from the origin `http://example.com`, if `example.com` is an authorized origin, the server would send back this response:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.com
```

An any-origin response would also permit this request, i.e.:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
```

Permitting user credentials or session

Allowing user credentials or the user's session to be sent with a CORS request allows the server to persist user data across CORS requests. This is useful if the server needs to check if the user is logged in before providing data (for example, only performing an action if a user is logged in - this would require the CORS request to be sent with credentials).

This can be achieved server-side for preflighted requests, by sending the `Access-Control-Allow-Credentials` header in response to the `OPTIONS` preflight request. Take the following case of a CORS request to `DELETE` a resource:

```
OPTIONS /cors HTTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: DELETE
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.org
Access-Control-Allow-Methods: DELETE
Access-Control-Allow-Credentials: true
```

The `Access-Control-Allow-Credentials: true` line indicates that the following `DELETE` CORS request may be sent with user credentials.

Preflighting requests

A basic CORS request is allowed to use one of only two methods:

- GET
- POST

and only a few select headers. POST CORS requests can additionally choose from only three content types.

To avoid this issue, requests that wish to use other methods, headers, or content types must first issue a *preflight* request, which is an `OPTIONS` request that includes access-control Request headers. For example, this is a preflight request that checks if the server will accept a `PUT` request that includes a `DNT` header:

```
OPTIONS /cors HTTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: DNT
```

Server: responding to preflight requests

When a server receives a preflight request, it must check if it supports the requested method and headers, and send back a response that indicates its ability to support the request, as well as any other permitted data (such as credentials).

These are indicated in access-control Allow headers. The server may also send back an access-

control `Max-Age` header, indicating how long the preflight response can be cached for.

This is what a request-response cycle for a preflight request might look like:

```
OPTIONS /cors HHTP/1.1
Host: example.com
Origin: example.org
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: DNT
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: example.org
Access-Control-Allow-Methods: PUT
Access-Control-Allow-Headers: DNT
```

Read Cross Origin and Access Control online: https://riptutorial.com/http/topic/3424/cross-origin-and-access-control

Chapter 5: HTTP for APIs

Remarks

HTTP APIs use a wide spectrum of HTTP verbs and typically return JSON or XML responses.

Examples

Create a resource

Not everyone agrees on what the most semantically correct method for resource creation is. Thus, your API could accept `POST` or `PUT` requests, or either.

The server should respond with `201 Created` if the resource was successfully created. Pick the most appropriate error code if it was not.

For example, if you provide an API to create employee records, the request/response might look like this:

```
POST /employees HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "name": "Charlie Smith",
    "age": 38,
    "job_title": "Software Developer",
    "salary": 54895.00
}
```

```
HTTP/1.1 201 Created
Location: /employees/1/charlie-smith
Content-Type: application/json

{
    "employee": {
        "name": "Charlie Smith",
        "age": 38,
        "job_title": "Software Developer",
        "salary": 54895.00
        "links": [
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "self",
                "method": "GET"
            },
            {
                "uri": "/employees/1/charlie-smith",
                "rel": "delete",
                "method": "DELETE"
            },
            {
```

```
                "uri": "/employees/1/charlie-smith",
                "rel": "edit",
                "method": "PATCH"
            }
        ]
    },
    "links": [
        {
            "uri": "/employees",
            "rel": "create",
            "method": "POST"
        }
    ]
}
```

Including the `links` JSON fields in the response enables the client to access resource related to the new resource and to the application as a whole, without having to know their URIs or methods beforehand.

Edit a resource

Editing or updating a resource is a common purpose for APIs. Edits can be achieved by sending either `POST`, `PUT` or `PATCH` requests to the respective resource. Although `POST` is allowed to append data to a resource's existing representation it is recommended to use either `PUT` or `PATCH` as they convey a more explicit semantic.

Your server should respond with `200 OK` if the update was performed, or `202 Accepted` if it has yet to be applied. Pick the most appropriate error code if it cannot be completed.

Full updates

`PUT` has the semantics of replacing the current representation with the payload included in the request. If the payload is not of the same representation type as the current representation of the resource to update, the server can decide which approach to take. RFC7231 defines that the server can either

- Reconfigure the target resource to reflect the new media type
- Transform the PUT representation to a format consistent with that of the resouce before saving it as the new resource state
- Reject the request with a `415 Unsupported Media Type` response indicating that the target resource is limited to a specific (set) of media types.

A base resource containing a JSON HAL representation like ...

```
{
    "name": "Charlie Smith",
    "age": 39,
    "job_title": "Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
```

```
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    }
}
```

... may receive an update request like this

```
PUT /users/1234 HTTP/1.1
Host: http://www.acmee.com
Content-Type: "application/json; charset=utf-8"
Content-Length: 85
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer"
}
```

The server may now replace the state of the resource with the given request body and also change the content-type from `application/hal+json` to `application/json` or convert the JSON payload to a JSON HAL representation and then replace the content of the resource with the transformed one or reject the update request due to an inaplicable media type with a `415 Unsupported Media Type` response.

There is a difference between replacing the content directly or first transforming the representation to the defined representation model and then replacing the existing content with the transformed one. A subsequent `GET` request will return the following response on a direct replacement:

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer"
}
```

while the transformation and then replace approach will return the following representation:

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: e0023aa4e
```

```
{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    }
}
```

Side-Effects

Note that `PUT` is allowed to have side-effects although it is defined as idempotent operation! This is documented in RFC7231 as

> A PUT request applied to the target resource **can have side effects on other resources**. For example, an article might have a URI for identifying "the current version" (a resource) that is separate from the URIs identifying each particular version (different resources that at one point shared the same state as the current version resource). A successful PUT request on "the current version" URI might therefore create a new version resource in addition to changing the state of the target resource, and might also cause links to be added between the related resources.

Producing additional log entries is not considered as side effect usually as this is certainly no state of a resource in general.

Partial updates

RFC7231 mentions this regarding partial updates:

> Partial content updates are possible by targeting a separately identified resource with state that overlaps a portion of the larger resource, or by using a different method that has been specifically defined for partial updates (for example, the PATCH method defined in RFC5789).

Partial updates can therefore be performed in two flavors:

- Have a resource embed multiple smaller sub-resources and update only a respective sub-resource instead of the full resource via `PUT`
- Using `PATCH` and instruct the server what to update

Partial update with overlapping state

If a user representation needs to be partially updated due to a move of a user to an other location, instead of updating the user directly, the related resource should be updated directly which reflects to a partial update of the user representation.

Before the move a user had the following representation

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept: application/hal+json; charset=utf-8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    },
    "_embedded": {
        "ea:address": {
            "street": "Terrace Drive, Central Park",
            "zip": "NY 10024"
            "city": "New York",
            "country": "United States of America",
            "_links": {
                "self": { "href": "/address/abc" },
                "google_maps": { "href": "http://maps.google.com/?ll=40.7739166,-73.970176" }
            }
        }
    }
}
```

As the user is moving to a new location she updates her location information like this:

```
PUT /address/abc HTTP/1.1
Host: http://www.acmee.com
Content-Type: "application/json; charset=utf-8"
Content-Length: 109
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)

{
    "street": "Standford Ave",
    "zip": "CA 94306",
    "city": "Pablo Alto",
    "country": "United States of America"
}
```

With the transformation-before-replace semantic for the mismatched media-type between the

existing address resource and the one in the request, as described above, the address resource is now updated which has the effect that on a subsequent `GET` request on the user resource the new address for the user is returned.

```
GET /users/1234 HTTP/1.1
Host: http://www.acmee.com
Accept: application/hal+json; charset=utf-8
Accept-Encoding: gzip, deflate
Accept-Language: en-us
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
ETag: "e0023aa4e"

{
    "name": "Charlie Gold-Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "_links": {
        "self": { "href": "/users/1234" },
        "employee": { "href": "http://www.acmee.com" },
        "curies": [{ "name": "ea", "href": "http://www.acmee.com/docs/rels/{rel}", templated":
true}],
        "ea:admin": [
            "href": "/admin/2",
            "title": "Admin"
        ]
    },
    "_embedded": {
        "ea:address": {
            "street": "Standford Ave",
            "zip": "CA 94306",
            "city": "Pablo Alto",
            "country": "United States of America"
            "_links": {
                "self": { "href": "/address/abc" },
                "google_maps": { "href": "http://maps.google.com/?ll=37.4241311,-122.1524475"
}
            }
        }
    }
}
```

Patching partial data

`PATCH` is defined in RFC5789 and is not directly part of the HTTP spec per se. A common misbelief is, that sending only the fields that should be partially updated is enough within a `PATCH` request. The specification therefore states

> The PATCH method requests that a set of changes described in the request entity be applied to the resource identified by the Request-URI. The set of changes is represented in a format called a "patch document" identified by a media type.

This means that a client should calculate the necessary steps needed to transform the resource from state A to state B and send these instructions to the server.

A popular JSON based media-type for patching is JSON Patch.

If the age and the job-title of our sample user changes and an additional field representing the income of the user should be added a partial update using `PATCH` using JSON Patch may look like this:

```
PATCH /users/1234 HTTP/1.1
Host: http://www.acmee.com
Content-Type: application/json-patch+json; charset=utf-8
Content-Length: 188
Accept: application/json
If-Match: "e0023aa4e"

[
    { "op": "replace", "path": "/age", "value": 40 },
    { "op": "replace", "path": "/job_title", "value": "Senior Software Developer" },
    { "op": "add", "path": "/salery", "value": 63985.00 }
]
```

`PATCH` may update multiple resources at once and requires to apply the changes atomically which means either all changes have to be applied or none at all which puts transactional burden on the API implementor.

A successful update may return something like this

```
HTTP/1.1 200 OK
Location: /users/1234
Content-Type: application/json
ETag: "df00eb258"

{
    "name": "Charlie Smith",
    "age": 40,
    "job_title": "Senior Software Developer",
    "salary": 63985.00
}
```

though is not restricted to `200 OK` response codes only.

To prevent in-between updates (changes done in-between the previous fetch of the representation state and the update) `ETag`, `If-Match` or `If-Unmodified-Since` header should be used.

Error Handling

The spec on `PATCH` recommends the following error handling:

Type	Error Code
Malformed patch document	`400 Bad Request`
Unsupported patch document	`415 Unsupported Media Type`
Unprocessable request, i.e. if the resoure would become invalid by applying the patch	`422 Unprocessable Entity`

Type	Error Code
Resource not found	`404 Not Found`
Conflicting state, i.e. a rename (move) of a field which does not exist	`409 Conflict`
Conflicting modification, i.e. if the client uses a `If-Match` or `If-Unmodified-Since` header which validation failed. If no precondition was available, the latter error code should be returned	`412 Precondition Failed` or `409 Conflict`
Concurrent modification, i.e. if the request needs to be applied before acception further `PATCH` requests	`409 Conflict`

Delete a resource

Another common use of HTTP APIs is to delete an existing resource. This should usually be done using `DELETE` requests.

If the deletion was successful, the server should return `200 OK`; an appropriate error code if it was not.

If our employee Charlie Smith has left the company and we now want to delete his records, that might look like this:

```
DELETE /employees/1/charlie-smith HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    'links': [
        {
            'uri': '/employees',
            'rel': 'create',
            'method': 'POST'
        }
    ]
}
```

List resources

The last common use of HTTP APIs is to obtain a list of existing resources on the server. Lists like this should be obtained using `GET` requests, since they only *retrieve* data.

The server should return `200 OK` if it can supply the list, or an appropriate error code if not.

Listing our employees, then, might look like this:

```
GET /employees HTTP/1.1
Host: example.com
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    'employees': [
        {
            'name': 'Charlie Smith',
            'age': 39,
            'job_title': 'Software Developer',
            'salary': 63985.00
            'links': [
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'self',
                    'method': 'GET'
                },
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'delete',
                    'method': 'DELETE'
                },
                {
                    'uri': '/employees/1/charlie-smith',
                    'rel': 'edit',
                    'method': 'PATCH'
                }
            ]
        },
        {
            'name': 'Donna Prima',
            'age': 30,
            'job_title': 'QA Tester',
            'salary': 77095.00
            'links': [
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'self',
                    'method': 'GET'
                },
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'delete',
                    'method': 'DELETE'
                },
                {
                    'uri': '/employees/2/donna-prima',
                    'rel': 'edit',
                    'method': 'PATCH'
                }
            ]
        }
    ],
    'links': [
        {
            'uri': '/employees/new',
            'rel': 'create',
            'method': 'PUT'
        }
    ]
}
```

Read HTTP for APIs online: https://riptutorial.com/http/topic/3423/http-for-apis

Chapter 6: HTTP requests

Parameters

HTTP Method	Purpose
`OPTIONS`	Retrieve information about the communication options (available methods and headers) available on the specified request URI.
`GET`	Retrieve the data identified by the request URI, or the data produced by the script available at the request URI.
`HEAD`	Identical to `GET` except that no message body will be returned by the server: only headers.
`POST`	Submit a block of data (specified in the message body) to the server for addition to the resouce specified in the request URI. Most commonly used for form processing.
`PUT`	Store the enclosed information (in the message body) as a new or updated resource under the given request URI.
`DELETE`	Delete, or queue for deletion, the resource identified by the request URI.
`TRACE`	Essentially an echo command: a functioning, compliant HTTP server must send the entire request back as the body of a 200 (OK) response.

Remarks

The `CONNECT` method is reserved by the method definitions specification for use with proxies that are able to switch between proxying and tunneling modes (such as for SSL tunneling).

Examples

Sending a minimal HTTP request manually using Telnet

This example demonstrates that HTTP is a text-based Internet communications protocol, and shows a basic HTTP request and the corresponding HTTP response.

You can use Telnet to manually send a minimal HTTP request from the command line, as follows.

1. Start a Telnet session to the web server `www.example.org` on port 80:

```
telnet www.example.org 80
```

Telnet reports that you have connected to the server:

```
Connected to www.example.org.
Escape character is '^]'.
```

2. Enter a request line to send a GET request URL path `/`, using HTTP 1.1

```
GET / HTTP/1.1
```

3. Enter an HTTP header field line to identify the host name part of the required URL, which is required in HTTP 1.1

```
Host: www.example.org
```

4. Enter a blank line to complete the request.

   The web server sends the HTTP response, which appears in the Telnet session.

The complete session, is as follows. The first line of the response is the *HTTP status line*, which includes the status code 200 and the status text *OK*, which indicate that the request was processed successfully. This is followed by a number of HTTP header fields, a blank line, and the HTML response.

```
$ telnet www.example.org 80
Trying 2606:2800:220:1:248:1893:25c8:1946...
Connected to www.example.org.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=604800
Content-Type: text/html
Date: Thu, 21 Jul 2016 15:56:05 GMT
Etag: "359670651"
Expires: Thu, 28 Jul 2016 15:56:05 GMT
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
Server: ECS (lga/1318)
Vary: Accept-Encoding
X-Cache: HIT
x-ec-custom-error: 1
Content-Length: 1270

<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
</head>
<body>
<div>
    <h1>Example Domain</h1>
```

```
    <p>This domain is established to be used for illustrative examples in documents. You may
use this
    domain in examples without prior coordination or asking for permission.</p>
    <p><a href="http://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
```

(`style` element and blank lines removed from the HTML reponse, for brevity.)

Basic request format

In HTTP 1.1, a minimal HTTP request consists of a request line and a `Host` header:

```
GET /search HTTP/1.1 \r\n
Host: google.com \r\n
\r\n
```

The first line has this format:

```
Method Request-URI HTTP-Version CRLF
```

`Method` should be a valid HTTP method; one of [1][2]:

- `OPTIONS`
- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `PATCH`
- `TRACE`
- `CONNECT`

`Request-URI` indicates either the URI or the path to the resource that the client is requesting. It can be either:

- a fully-qualified URI, including scheme, host, (optional) port and path; or
- a path, in which case the host must be specified in the `Host` header

`HTTP-Version` indicates the version of the HTTP protocol the client is using. For HTTP 1.1 requests this must always be `HTTP/1.1`.

The request line ends with a carriage return—line feed pair, usually represented by `\r\n`.

Request header fields

Header fields (usually just called ‘headers’) may be added to an HTTP request to provide additional information with the request. A header has semantics similar to parameters passed to a method in any programming language that supports such things.

A request with `Host`, `User-Agent` and `Referer` headers might look like this:

```
GET /search HTTP/1.1 \r\n
Host: google.com \r\n
User-Agent: Chrome/54.0.2803.1 \r\n
Referer: http://google.com/ \r\n
\r\n
```

A full list of supported HTTP 1.1 request headers can be found in the specification. The most common are:

- `Host` - the host name part of the request URL (required in HTTP/1.1)
- `User-Agent` - a string that represents the user agent requesting;
- `Referer` - the URI that the client was referred here from; and
- `If-Modified-Since` - gives a date that the server can use to determine if a resource has changed and indicate that the client can used a cached copy if it has not.

A header should be formed as `Name: Value CRLF`. `Name` is the header name, such as `User-Agent`. `Value` is the data assigned to it, and the line should end with a CRLF. Header names are case-insensitive and may only use letters, digits and the characters `!#$%&'*+-.^_`|~` (RFC7230 section 3.2.6 Field value components).

The `Referer` header field name is a typo for 'referrer', introduced accidentally in RFC1945.

Message bodies

Some HTTP requests may contain a message body. This is additional data that the server will use to process the request. Message bodies are most often used in POST or PATCH and PUT requests, to provide new data that the server should apply to a resource.

Requests that include a message body should always include its length in bytes with `Content-Length` header.

A message body is included *after* all headers and a double CRLF. An example PUT request with a body might look like this:

```
PUT /files/129742 HTTP/1.1\r\n
Host: example.com\r\n
User-Agent: Chrome/54.0.2803.1\r\n
Content-Length: 202\r\n
\r\n
This is a message body. All content in this message body should be stored under the
/files/129742 path, as specified by the PUT specification. The message body does
not have to be terminated with CRLF.
```

`HEAD` and `TRACE` requests must not include a message body.

Read HTTP requests online: https://riptutorial.com/http/topic/2909/http-requests

Chapter 7: HTTP responses

Parameters

Status Code	Reason-Phrase — Description
100	**Continue** — the client should send the following part of a multi-part request.
101	**Switching Protocols** — the server is changing the version or type of protocol used in this communication.
200	**OK** — the server has received and completed the client's request.
201	**Created** — the server has accepted the request and created a new resource, which is available under the URI in the `Location` header.
202	**Accepted** — the server has received and accepted the client's request, but it has not yet started or completed processing.
203	**Non-Authoritative Information** — the server is returning data that may be a sub- or superset of the information available on the original server. Mainly used by proxies.
204	**No Content** — used in place of 200 (OK) when there is no body to the response.
205	**Reset Content** — identical to 204 (No Content), but the client should reload the active document view.
206	**Partial Content** — used in place of 200 (OK) when the client requested a `Range` header.
300	**Multiple Choices** — the requested resource is available at multiple URIs, and the client should redirect the request to a URI specified in the list in the message body.
301	**Moved Permanently** — the requested resource is no longer available at this URI, and the client should redirect this and all future requests to the URI specified in the `Location` header.
302	**Found** — the resource temporarily resides under a different URI. This request should be redirected on user confirmation to the URI in the `Location` header, but future requests should not be altered.
303	**See Other** — very similar to 302 (Found), but does not require user input to redirect to the provided URI. The provided URI should be retrieved with a GET request.

Status Code	Reason-Phrase — Description
304	**Not Modified** — the client sent an `If-Modified-Since` or similar header, and the resource has not been modified since that point; the client should display a cached copy of the resource.
305	**Use Proxy** — the requested resource must be requested again through the proxy specified in the `Location` header field.
307	**Temporary Redirect** — identical to 302 (Found), but HTTP 1.0 clients do not support 307 responses.
400	**Bad Request** — the client sent a malformed request containing syntax errors, and should modify the request to correct this before repeating it.
401	**Unauthorized** — the requested resource is not available without authentication. The client may repeat the request using an `Authorization` header to provide authentication details.
402	**Payment Required** — reserved, unspecified status code for use by applications that require user subscriptions to view content.
403	**Forbidden** — the server understands the request, but refuses to fulfil it due to existing access controls. The request should not be repeated.
404	**Not Found** — there is no resource available on this server that matches the requested URI. May be used in place of 403 to avoid exposing access control details.
405	**Method Not Allowed** — the resource does not support the request method (HTTP verb); the `Allow` header lists acceptable request methods.
406	**Not Acceptable** — the resource has characteristics that violate the accept headers sent in the request.
407	**Proxy Authentication Required** — similar to 401 (Unauthorized), but indicates that the client must first authenticate with the intermediate proxy.
408	**Request Timeout** — the server expected another request from the client, but none were provided within an acceptable timeframe.
409	**Conflict** — the request could not be completed because it conflicted with the current state of the resource.
410	**Gone** — similar to 404 (Not Found), but indicates a permanent removal. No forwarding address is available.
411	**Length Required** — the client did not specify a valid `Content-Length` header, and must do so before the server will accept this request.

Status Code	Reason-Phrase — Description
412	**Precondition Failed** — the resource is not available with all the conditions specified by the conditional headers sent by the client.
413	**Request Entity Too Large** — the server is presently unable to process a message body of the length that the client sent.
414	**Request-URI Too Long** — the server is refusing the request because the Request-URI is longer than the server is willing to interpret.
415	**Unsupported Media Type** — the server does not support the MIME or media type specified by the client, and cannot service this request.
416	**Requested Range Not Satisfiable** — the client requested a range of bytes, but the server cannot provide content to that specification.
417	**Expectation Failed** — the client specified constraints in the `Expect` header that the server cannot meet.
500	**Internal Server Error** — the server met an unexpected condition or error which prevents it from completing this request.
501	**Not Implemented** — the server does not support the functionality required to complete the request. Usually used to indicate a request method that is not supported on *any* resource.
502	**Bad Gateway** — the server is a proxy, and received an invalid response from the upstream server while processing this request.
503	**Service Unavailable** — the server is under high load or undergoing maintenance, and does not have the capacity to serve this request at present.
504	**Gateway Timeout** — the server is a proxy, and did not receive a response from the upstream server in a timely manner.
505	**HTTP Version Not Supported** — the server does not support the version of the HTTP protocol that the client made its request with.

Examples

Basic response format

When an HTTP server receives a well-formed HTTP request, it must process the information that request contains and return a response to the client. A simple HTTP 1.1 response, may look like any of the following, usually followed by a number of header fields, and possibly a response body:

```
HTTP/1.1 200 OK \r\n
```

```
HTTP/1.1 404 Not Found \r\n
```

```
HTTP/1.1 503 Service Unavailable \r\n
```

A simple HTTP 1.1 response has this format:

```
HTTP-Version Status-Code Reason-Phrase CRLF
```

As in a request, `HTTP-Version` indicates the version of the HTTP protocol in use; for HTTP 1.1 this must always be the string `HTTP/1.1`.

`Status-Code` is a three-digit code that indicates the status of the client's request. The first digit of this code is the *status class*, which places the status code into one of 5 categories of response [1]:

- `1xx` **Informational** - the server has received the request and processing is continuing
- `2xx` **Success** - the server has accepted and processed the request
- `3xx` **Redirection** - further action is necessary on the client's part to complete the request
- `4xx` **Client Errors** - the client sent a request that was malformed or cannot be fulfilled
- `5xx` **Server Errors** - the request was valid, but the server cannot fulfil it at present

`Reason-Phrase` is a short description of the status code. For example, code `200` has a reason phrase of `OK`; code `404` has a phrase of `Not Found`. A full list of reason phrases is available in Parameters, below, or in the HTTP specification.

The line ends with a carriage return—line feed pair, usually represented by `\r\n`.

Additional Headers

Like an HTTP request, an HTTP response may include additional headers to modify or augment the response it provides.

A full list of available headers is defined in §6.2 of the specification. The most commonly-used headers are:

- `Server`, which functions like a `User-Agent` request header for the server;
- `Location`, which is used on 201 and 3xx status responses to indicate a URI to redirect to; and
- `ETag`, which is a unique identifier for this version of the returned resource to enable clients to cache the response.

Response headers come after the status line, and as with request headers are formed as such:

```
Name: Value CRLF
```

`Name` provides the header name, such as `ETag` or `Location`, and `Value` provides the value that the server is setting for that header. The line ends with a CRLF.

A response with headers might look like this:

```
HTTP/1.1 201 Created \r\n
Server: WEBrick/1.3.1 \r\n
Location: http://example.com/files/129742 \r\n
```

Message Bodies

As with request bodies, HTTP responses may contain a message body. This provides additional data that the client will process. Notably, 200 OK responses to a well-formed GET request should always provide a message body containing the requested data. (If there is none, 204 No Content is a more appropriate response).

A message body is included after all headers and a double CRLF. As for requests, its length in bytes should be given with `Content-Length` header. A successful response to a GET request, therefore, might look like this:

```
HTTP/1.1 200 OK\r\n
Server: WEBrick/1.3.1\r\n
Content-Length: 39\r\n
ETag: 4f7e2ed02b836f60716a7a3227e2b5bda7ee12c53be282a5459d7851c2b4fdfd\r\n
\r\n
Nobody expects the Spanish Inquisition.
```

Read HTTP responses online: https://riptutorial.com/http/topic/3077/http-responses

Chapter 8: HTTP Status Codes

Introduction

In HTTP, status codes are a machine-readable mechanism indicating the result of a previously issued request. From RFC 7231, sec. 6: "The status-code element is a three-digit integer code giving the result of the attempt to understand and satisfy the request."

The formal grammar allows codes to be anything between `000` and `999`. However, only the range from `100` to `599` has assigned meaning.

Remarks

HTTP/1.1 defines a number of numeric HTTP status codes that appear in the status line - the first line of an HTTP response - to summarise what the client should do with the response.

The first digit of a status codes defines the response's class:

- `1xx` Informational
- `2xx` Client request successful
- `3xx` Request redirected - further action necessary, such as a new request
- `4xx` Client error - do not repeat the same request
- `5xx` Server error - maybe try again

In practice, it is not always easy to choose the most appropriate status code.

Examples

500 Internal Server Error

A **HTTP 500 Internal Server Error** is a general message meaning that the server encountered something unexpected. Applications (or the overarching web server) should use a 500 when there's an error processing the request - i.e. an exception is thrown, or a condition of the resource prevents the process completing.

Example status line:

```
HTTP/1.1 500 Internal Server Error
```

404 Not Found

HTTP 404 Not Found means that the server couldn't find the path using the URI that the client requested.

```
HTTP/1.1 404 Not Found
```

Most often, the requested file was deleted, but sometimes it can be a document root misconfiguration or a lack of permissions (though missing permissions more frequently triggers HTTP 403 Forbidden).

For example, Microsoft's IIS writes 404.0 (0 is the sub-status) to its log files when the requested file was deleted. But when the incoming request is blocked by request filtering rules, it writes 404.5-404.19 to log files according to which rule blocks the request. A more detailed error code reference can be found at Microsoft Support.

Denying access to protected files

Use 403 Forbidden when a client has requested a resource that is inaccessible due to existing access controls. For example, if your app has an `/admin` route that should only be accessible to users with administrative rights, you can use 403 when a normal user requests the page.

```
GET /admin HTTP/1.1
Host: example.com
```

```
HTTP/1.1 403 Forbidden
```

Successful request

Send an HTTP response with status code `200` to indicate a successful request. The HTTP response status line is then:

```
HTTP/1.1 200 OK
```

The status text `OK` is only informative. The response body (message payload) should contain a representation of the requested resource. If there is no representation 201 No Content should be used.

Responding to a conditional request for cached content

Send a **304 Not Modified** response status from the server send in response to a client request that contains headers `If-Modified-Since` and `If-None-Match`, if the request resource hasn't changed.

For example if a client request for a web page includes the header `If-Modified-Since: Fri, 22 Jul 2016 14:34:40 GMT` and the page wasn't modified since then, respond with the status line `HTTP/1.1 304 Not Modified`.

Top 10 HTTP Status Code

2xx Success

- **200 OK** - Standard response for successful HTTP requests.
- **201 Created** - The request has been fulfilled, resulting in the creation of a new resource.

- **204 No Content** - The server successfully processed the request and is not returning any content.

3xx Redirection

- **304 Not Modified** - Indicates that the resource has not been modified since the version specified by the request headers `If-Modified-Since` or `If-None-Match`.

4xx Client Error

- **400 Bad Request** - The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, too large size, invalid request message framing, or deceptive request routing).
- **401 Unauthorized** - *Similar to 403 Forbidden*, but specifically for use when authentication is required and has failed or has not yet been provided. The response must include a `WWW-Authenticate` header field containing a challenge applicable to the requested resource.
- **403 Forbidden** - The request was a valid request, but the server is refusing to respond to it. The user might be logged in but does not have the necessary permissions for the resource.
- **404 Not Found** - The requested resource could not be found but may be available in the future. Subsequent requests by the client are permissible.
- **409 Conflict** - Indicates that the request could not be processed because of conflict in the request, such as an edit conflict between multiple simultaneous updates.

5xx Server Error

- **500 Internal Server Error** - A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

Read HTTP Status Codes online: https://riptutorial.com/http/topic/2577/http-status-codes

Chapter 9: Response encodings and compression

Examples

HTTP compression

The HTTP message body can be compressed (since HTTP/1.1). Either by the server compresses the request and adds a `Content-Encoding` header, or by a proxy does and adds a `Transfer-Encoding` header.

A client may send an `Accept-Encoding` request header to indicate which encodings it accepts.

The most commonly used encodings are:

- gzip - deflate algorithm (LZ77) with CRC32 checksum implemented in "gzip" file's compression program (RFC1952)
- deflate - "zlib" data format (RFC1950), deflate algorithm (hybrid LZ77 and Huffman) with Adler32 checksum

Multiple compression methods

It is possible to compress an HTTP response message body more than once. The encoding names should then be separated by a comma in the order in which they were applied. For example, if a message has been compressed via deflate and then gzip, the header should look like:

```
Content-Encoding: deflate, gzip
```

Multiple `Content-Encoding` headers are also valid, though not recommended:

```
Content-Encoding: deflate
Content-Encoding: gzip
```

gzip compression

The client first sends a request with an `Accept-Encoding` header that indicates it supports gzip:

```
GET / HTTP/1.1\r\n
Host: www.google.com\r\n
Accept-Encoding: gzip, deflate\r\n
\r\n
```

The server may then send a response with a compressed response body and a `Content-Encoding` header that specifies that gzip encoding was used::

```
HTTP/1.1 200 OK\r\n
Content-Encoding: gzip\r\n
Content-Length: XX\r\n
\r\n
... compressed content ...
```

Read Response encodings and compression online:
https://riptutorial.com/http/topic/5046/response-encodings-and-compression

Credits

S. No	Chapters	Contributors
1	Getting started with HTTP	Community, DaSourcerer, Kornel, Peter Hilton
2	Authentication	DaSourcerer, Peter Hilton, Stefan Kögl
3	Caching HTTP responses	DaSourcerer, Kornel
4	Cross Origin and Access Control	ArtOfCode
5	HTTP for APIs	ArtOfCode, mnoronha, Peter Hilton, Roman Vottner
6	HTTP requests	artem, ArtOfCode, Jeff Bencteux, Peter Hilton
7	HTTP responses	ArtOfCode, Jeff Bencteux, Peter Hilton
8	HTTP Status Codes	ArtOfCode, DaSourcerer, Deltik, Kornel, Lex Li, mnoronha, Peter Hilton, Rptk99, Sender, Xevaquor
9	Response encodings and compression	Jeff Bencteux, Peter Hilton