Internet Engineering Task Force (IETF)
Request for Comments: 9111
Obsoletes: 7234
STD: 98
Category: Standards Track
ISSN: 2070-1721

R. Fielding, Editor
Adobe
M. Nottingham, Editor
Fastly
J. Reschke, Editor
greenbytes

June 2022

HTTP Caching

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines HTTP caches and the associated header fields that control cache behavior or indicate cacheable response messages.

This document obsoletes RFC 7234.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841[1].

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9111[2].

Copyright Notice

[1] https://www.rfc-editor.org/rfc/rfc7841.html#section-2
[2] https://www.rfc-editor.org/info/rfc9111
[3] https://trustee.ietf.org/license-info

created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction

The Hypertext Transfer Protocol (HTTP) is a stateless application-level request/response protocol that uses extensible semantics and self-descriptive messages for flexible interaction with network-based hypertext information systems. It is typically used for distributed information systems, where the use of response caches can improve performance. This document defines aspects of HTTP related to caching and reusing response messages.

An HTTP *cache* is a local store of response messages and the subsystem that controls storage, retrieval, and deletion of messages in it. A cache stores cacheable responses to reduce the response time and network bandwidth consumption on future equivalent requests. Any client or server MAY use a cache, though not when acting as a tunnel (Section 3.7 of [HTTP]).

A *shared cache* is a cache that stores responses for reuse by more than one user; shared caches are usually (but not always) deployed as a part of an intermediary. A *private cache*, in contrast, is dedicated to a single user; often, they are deployed as a component of a user agent.

The goal of HTTP caching is significantly improving performance by reusing a prior response message to satisfy a current request. A cache considers a stored response "fresh", as defined in Section 4.2, if it can be reused without "validation" (checking with the origin server to see if the cached response remains valid for this request). A fresh response can therefore reduce both latency and network overhead each time the cache reuses it. When a cached response is not fresh, it might still be reusable if validation can freshen it (Section 4.3) or if the origin is unavailable (Section 4.2.4).

This document obsoletes RFC 7234, with the changes being summarized in Appendix B.

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Section 2 of [HTTP] defines conformance criteria and contains considerations regarding error handling.

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234], extended with the notation for case-sensitivity in strings defined in [RFC7405].

It also uses a list extension, defined in Section 5.6.1 of [HTTP], that allows for compact definition of comma-separated lists using a "#" operator (similar to how the "*" operator indicates repetition). Appendix A shows the collected grammar with all list operators expanded to standard ABNF notation.

1.2.1. Imported Rules

The following core rule is included by reference, as defined in [RFC5234], Appendix B.1: DIGIT (decimal 0-9).

[HTTP] defines the following rules:

```
HTTP-date     = <HTTP-date, see [HTTP], Section 5.6.7>
OWS           = <OWS, see [HTTP], Section 5.6.3>
field-name    = <field-name, see [HTTP], Section 5.1>
quoted-string = <quoted-string, see [HTTP], Section 5.6.4>
token         = <token, see [HTTP], Section 5.6.2>
```

1.2.2. Delta Seconds

The delta-seconds rule specifies a non-negative integer, representing time in seconds.

```
delta-seconds  = 1*DIGIT
```

A recipient parsing a delta-seconds value and converting it to binary form ought to use an arithmetic type of at least 31 bits of non-negative integer range. If a cache receives a delta-seconds value greater than the greatest integer it can represent, or if any of its subsequent calculations overflows, the cache MUST consider the value to be 2147483648 (2^{31}) or the greatest positive integer it can conveniently represent.

> **Note:** The value 2147483648 is here for historical reasons, represents infinity (over 68 years), and does not need to be stored in binary form; an implementation could produce it as a string if any overflow occurs, even if the calculations are performed with an arithmetic type incapable of directly representing that number. What matters here is that an overflow be detected and not treated as a negative value in later calculations.

2. Overview of Cache Operation

Proper cache operation preserves the semantics of HTTP transfers while reducing the transmission of information already held in the cache. See Section 3 of [HTTP] for the general terminology and core concepts of HTTP.

Although caching is an entirely OPTIONAL feature of HTTP, it can be assumed that reusing a cached response is desirable and that such reuse is the default behavior when no requirement or local configuration prevents it. Therefore, HTTP cache requirements are focused on preventing a cache from either storing a non-reusable response or reusing a stored response inappropriately, rather than mandating that caches always store and reuse particular responses.

The *cache key* is the information a cache uses to choose a response and is composed from, at a minimum, the request method and target URI used to retrieve the stored response; the method determines under which circumstances that response can be used to satisfy a subsequent request. However, many HTTP caches in common use today only cache GET responses and therefore only use the URI as the cache key.

A cache might store multiple responses for a request target that is subject to content negotiation. Caches differentiate these responses by incorporating some of the original request's header fields into the cache key as well, using information in the Vary response header field, as per Section 4.1.

Caches might incorporate additional material into the cache key. For example, user agent caches might include the referring site's identity, thereby "double keying" the cache to avoid some privacy risks (see Section 7.2).

Most commonly, caches store the successful result of a retrieval request: i.e., a 200 (OK) response to a GET request, which contains a representation of the target resource (Section 9.3.1 of [HTTP]). However, it is also possible to store redirects, negative results (e.g., 404 (Not Found)), incomplete results (e.g., 206 (Partial Content)), and responses to methods other than GET if the method's definition allows such caching and defines something suitable for use as a cache key.

A cache is *disconnected* when it cannot contact the origin server or otherwise find a forward path for a request. A disconnected cache can serve stale responses in some circumstances (Section 4.2.4).

3. Storing Responses in Caches

A cache MUST NOT store a response to a request unless:

- the request method is understood by the cache;
- the response status code is final (see Section 15 of [HTTP]);
- if the response status code is 206 or 304, or the must-understand cache directive (see Section 5.2.2.3) is present: the cache understands the response status code;
- the no-store cache directive is not present in the response (see Section 5.2.2.5);
- if the cache is shared: the private response directive is either not present or allows a shared cache to store a modified response; see Section 5.2.2.7);
- if the cache is shared: the Authorization header field is not present in the request (see Section 11.6.2 of [HTTP]) or a response directive is present that explicitly allows shared caching (see Section 3.5); and
- the response contains at least one of the following:
  - a public response directive (see Section 5.2.2.9);
  - a private response directive, if the cache is not shared (see Section 5.2.2.7);
  - an Expires header field (see Section 5.3);
  - a max-age response directive (see Section 5.2.2.1);
  - if the cache is shared: an s-maxage response directive (see Section 5.2.2.10);
  - a cache extension that allows it to be cached (see Section 5.2.3); or
  - a status code that is defined as heuristically cacheable (see Section 4.2.2).

Note that a cache extension can override any of the requirements listed; see Section 5.2.3.

In this context, a cache has "understood" a request method or a response status code if it recognizes it and implements all specified caching-related behavior.

Note that, in normal operation, some caches will not store a response that has neither a cache validator nor an explicit expiration time, as such responses are not usually useful to store. However, caches are not prohibited from storing such responses.

3.1. Storing Header and Trailer Fields

Caches MUST include all received response header fields — including unrecognized ones — when storing a response; this assures that new HTTP header fields can be successfully deployed. However, the following exceptions are made:

- The Connection header field and fields whose names are listed in it are required by Section 7.6.1 of [HTTP] to be removed before forwarding the message. This MAY be implemented by doing so before storage.
- Likewise, some fields' semantics require them to be removed before forwarding the message, and this MAY be implemented by doing so before storage; see Section 7.6.1 of [HTTP] for some examples.
- The no-cache (Section 5.2.2.4) and private (Section 5.2.2.7) cache directives can have arguments that prevent storage of header fields by all caches and shared caches, respectively.
- Header fields that are specific to the proxy that a cache uses when forwarding a request MUST NOT be stored, unless the cache incorporates the identity of the proxy into the cache key. Effectively, this is limited to Proxy-Authenticate (Section 11.7.1 of [HTTP]), Proxy-Authentication-Info (Section 11.7.3 of [HTTP]), and Proxy-Authorization (Section 11.7.2 of [HTTP]).

Caches MAY either store trailer fields separate from header fields or discard them. Caches MUST NOT combine trailer fields with header fields.

3.2. Updating Stored Header Fields

Caches are required to update a stored response's header fields from another (typically newer) response in several situations; for example, see Sections 3.4, 4.3.4, and 4.3.5.

When doing so, the cache MUST add each header field in the provided response to the stored response, replacing field values that are already present, with the following exceptions:

- Header fields excepted from storage in Section 3.1,
- Header fields that the cache's stored response depends upon, as described below,
- Header fields that are automatically processed and removed by the recipient, as described below, and
- The Content-Length header field.

In some cases, caches (especially in user agents) store the results of processing the received response, rather than the response itself, and updating header fields that affect that processing can result in inconsistent behavior and security issues. Caches in this situation MAY omit these header fields from updating stored responses on an exceptional basis but SHOULD limit such omission to those fields necessary to assure integrity of the stored response.

For example, a browser might decode the content coding of a response while it is being received, creating a disconnect between the data it has stored and the response's original metadata. Updating that stored metadata with a different Content-Encoding header field would be problematic. Likewise, a browser might store a post-parse HTML tree rather than the content received in the response; updating the Content-Type header field would not be workable in this case because any assumptions about the format made in parsing would now be invalid.

Furthermore, some fields are automatically processed and removed by the HTTP implementation, such as the Content-Range header field. Implementations MAY automatically omit such header fields from updates, even when the processing does not actually occur.

Note that the Content-* prefix is not a signal that a header field is omitted from update; it is a convention for MIME header fields, not HTTP.

3.3. Storing Incomplete Responses

If the request method is GET, the response status code is 200 (OK), and the entire response header section has been received, a cache MAY store a response that is not complete (Section 6.1 of [HTTP]) provided that the stored response is recorded as being incomplete. Likewise, a 206 (Partial Content) response MAY be stored as if it were an incomplete 200 (OK) response. However, a cache MUST NOT store incomplete or partial-content responses if it does not support the Range and Content-Range header fields or if it does not understand the range units used in those fields.

A cache MAY complete a stored incomplete response by making a subsequent range request (Section 14.2 of [HTTP]) and combining the successful response with the stored response, as defined in Section 3.4. A cache MUST NOT use an incomplete response to answer requests unless the response has been made complete, or the request is partial and specifies a range wholly within the incomplete response. A cache MUST NOT send a partial response to a client without explicitly marking it using the 206 (Partial Content) status code.

3.4. Combining Partial Content

A response might transfer only a partial representation if the connection closed prematurely or if the request used one or more Range specifiers (Section 14.2 of [HTTP]). After several such transfers, a cache might have received several ranges of the same representation. A cache MAY combine these ranges into a single stored response, and reuse that response to satisfy later requests, if they all share the same strong validator and the cache complies with the client requirements in Section 15.3.7.3 of [HTTP].

When combining the new response with one or more stored responses, a cache MUST update the stored response header fields using the header fields provided in the new response, as per Section 3.2.

3.5. Storing Responses to Authenticated Requests

A shared cache MUST NOT use a cached response to a request with an Authorization header field (Section 11.6.2 of [HTTP]) to satisfy any subsequent request unless the response contains a Cache-Control field with a response directive (Section 5.2.2) that allows it to be stored by a shared cache, and the cache conforms to the requirements of that directive for that response.

In this specification, the following response directives have such an effect: must-revalidate (Section 5.2.2.2), public (Section 5.2.2.9), and s-maxage (Section 5.2.2.10).

4. Constructing Responses from Caches

When presented with a request, a cache MUST NOT reuse a stored response unless:

- the presented target URI (Section 7.1 of [HTTP]) and that of the stored response match, and
- the request method associated with the stored response allows it to be used for the presented request, and
- request header fields nominated by the stored response (if any) match those presented (see Section 4.1), and
- the stored response does not contain the no-cache directive (Section 5.2.2.4), unless it is successfully validated (Section 4.3), and
- the stored response is one of the following:
  - fresh (see Section 4.2), or
  - allowed to be served stale (see Section 4.2.4), or
  - successfully validated (see Section 4.3).

Note that a cache extension can override any of the requirements listed; see Section 5.2.3.

When a stored response is used to satisfy a request without validation, a cache MUST generate an Age header field (Section 5.1), replacing any present in the response with a value equal to the stored response's current_age; see Section 4.2.3.

A cache MUST write through requests with methods that are unsafe (Section 9.2.1 of [HTTP]) to the origin server; i.e., a cache is not allowed to generate a reply to such a request before having forwarded the request and having received a corresponding response.

Also, note that unsafe requests might invalidate already-stored responses; see Section 4.4.

A cache can use a response that is stored or storable to satisfy multiple requests, provided that it is allowed to reuse that response for the requests in question. This enables a cache to *collapse requests* — or combine multiple incoming requests into a single forward request upon a cache miss — thereby reducing load on the origin server and network. Note, however, that if the cache cannot use the returned response for some or all of the collapsed requests, it will need to forward the requests in order to satisfy them, potentially introducing additional latency.

When more than one suitable response is stored, a cache MUST use the most recent one (as determined by the Date header field). It can also forward the request with "Cache-Control: max-age=0" or "Cache-Control: no-cache" to disambiguate which response to use.

A cache without a clock (Section 5.6.7 of [HTTP]) MUST revalidate stored responses upon every use.

4.1. Calculating Cache Keys with the Vary Header Field

When a cache receives a request that can be satisfied by a stored response and that stored response contains a Vary header field (Section 12.5.5 of [HTTP]), the cache MUST NOT use that stored response without revalidation unless all the presented request header fields nominated by that Vary field value match those fields in the original request (i.e., the request that caused the cached response to be stored).

The header fields from two requests are defined to match if and only if those in the first request can be transformed to those in the second request by applying any of the following:

- adding or removing whitespace, where allowed in the header field's syntax
- combining multiple header field lines with the same field name (see Section 5.2 of [HTTP])
- normalizing both header field values in a way that is known to have identical semantics, according to the header field's specification (e.g., reordering field values when order is not significant; case-normalization, where values are defined to be case-insensitive)

If (after any normalization that might take place) a header field is absent from a request, it can only match another request if it is also absent there.

A stored response with a Vary header field value containing a member "*" always fails to match.

If multiple stored responses match, the cache will need to choose one to use. When a nominated request header field has a known mechanism for ranking preference (e.g., qvalues on Accept and similar request header fields), that mechanism MAY be used to choose a preferred response. If such a mechanism is not available, or leads to equally preferred responses, the most recent response (as determined by the Date header field) is chosen, as per Section 4.

Some resources mistakenly omit the Vary header field from their default response (i.e., the one sent when the request does not express any preferences), with the effect of choosing it for subsequent requests to that resource even when more preferable responses are available. When a cache has multiple stored responses for a target URI and one or more omits the Vary header field, the cache SHOULD choose the most recent (see Section 4.2.3) stored response with a valid Vary field value.

If no stored response matches, the cache cannot satisfy the presented request. Typically, the request is forwarded to the origin server, potentially with preconditions added to describe what responses the cache has already stored (Section 4.3).

4.2. Freshness

A *fresh* response is one whose age has not yet exceeded its freshness lifetime. Conversely, a *stale* response is one where it has.

A response's *freshness lifetime* is the length of time between its generation by the origin server and its expiration time. An *explicit expiration time* is the time at which the origin server intends that a stored response can no longer be used by a cache without further validation, whereas a *heuristic expiration time* is assigned by a cache when no explicit expiration time is available.

A response's *age* is the time that has passed since it was generated by, or successfully validated with, the origin server.

When a response is fresh, it can be used to satisfy subsequent requests without contacting the origin server, thereby improving efficiency.

The primary mechanism for determining freshness is for an origin server to provide an explicit expiration time in the future, using either the Expires header field (Section 5.3) or the max-age response directive (Section 5.2.2.1). Generally, origin servers will assign future explicit expiration times to responses in the belief that the representation is not likely to change in a semantically significant way before the expiration time is reached.

If an origin server wishes to force a cache to validate every request, it can assign an explicit expiration time in the past to indicate that the response is already stale. Compliant caches will normally validate a stale cached response before reusing it for subsequent requests (see Section 4.2.4).

Since origin servers do not always provide explicit expiration times, caches are also allowed to use a heuristic to determine an expiration time under certain circumstances (see Section 4.2.2).

The calculation to determine if a response is fresh is:

```
response_is_fresh = (freshness_lifetime > current_age)
```

freshness_lifetime is defined in Section 4.2.1; current_age is defined in Section 4.2.3.

Clients can send the max-age or min-fresh request directives (Section 5.2.1) to suggest limits on the freshness calculations for the corresponding response. However, caches are not required to honor them.

When calculating freshness, to avoid common problems in date parsing:

- Although all date formats are specified to be case-sensitive, a cache recipient SHOULD match the field value case-insensitively.

- If a cache recipient's internal implementation of time has less resolution than the value of an HTTP-date, the recipient MUST internally represent a parsed Expires date as the nearest time equal to or earlier than the received value.
- A cache recipient MUST NOT allow local time zones to influence the calculation or comparison of an age or expiration time.
- A cache recipient SHOULD consider a date with a zone abbreviation other than "GMT" to be invalid for calculating expiration.

Note that freshness applies only to cache operation; it cannot be used to force a user agent to refresh its display or reload a resource. See Section 6 for an explanation of the difference between caches and history mechanisms.

4.2.1. Calculating Freshness Lifetime

A cache can calculate the freshness lifetime (denoted as freshness_lifetime) of a response by evaluating the following rules and using the first match:

- If the cache is shared and the s-maxage response directive (Section 5.2.2.10) is present, use its value, or
- If the max-age response directive (Section 5.2.2.1) is present, use its value, or
- If the Expires response header field (Section 5.3) is present, use its value minus the value of the Date response header field (using the time the message was received if it is not present, as per Section 6.6.1 of [HTTP]), or
- Otherwise, no explicit expiration time is present in the response. A heuristic freshness lifetime might be applicable; see Section 4.2.2.

Note that this calculation is intended to reduce clock skew by using the clock information provided by the origin server whenever possible.

When there is more than one value present for a given directive (e.g., two Expires header field lines or multiple Cache-Control: max-age directives), either the first occurrence should be used or the response should be considered stale. If directives conflict (e.g., both max-age and no-cache are present), the most restrictive directive should be honored. Caches are encouraged to consider responses that have invalid freshness information (e.g., a max-age directive with non-integer content) to be stale.

4.2.2. Calculating Heuristic Freshness

Since origin servers do not always provide explicit expiration times, a cache MAY assign a heuristic expiration time when an explicit time is not specified, employing algorithms that use other field values (such as the Last-Modified time) to estimate a plausible expiration time. This specification does not provide specific algorithms, but it does impose worst-case constraints on their results.

A cache MUST NOT use heuristics to determine freshness when an explicit expiration time is present in the stored response. Because of the requirements in Section 3, heuristics can only be used on responses without explicit freshness whose status codes are defined as *heuristically cacheable* (e.g., see Section 15.1 of [HTTP]) and on responses without explicit freshness that have been marked as explicitly cacheable (e.g., with a public response directive).

Note that in previous specifications, heuristically cacheable response status codes were called "cacheable by default".

If the response has a Last-Modified header field (Section 8.8.2 of [HTTP]), caches are encouraged to use a heuristic expiration value that is no more than some fraction of the interval since that time. A typical setting of this fraction might be 10%.

> **Note:** A previous version of the HTTP specification (Section 13.9 of [RFC2616]) prohibited caches from calculating heuristic freshness for URIs with query components (i.e., those containing "?"). In practice, this has not been widely implemented. Therefore, origin servers are encouraged to send explicit directives (e.g., Cache-Control: no-cache) if they wish to prevent caching.

4.2.3. Calculating Age

The Age header field is used to convey an estimated age of the response message when obtained from a cache. The Age field value is the cache's estimate of the number of seconds since the origin server generated or validated the response. The Age value is therefore the sum of the time that the response has been resident in each of the caches along the path from the origin server, plus the time it has been in transit along network paths.

Age calculation uses the following data:

age_value The term "age_value" denotes the value of the Age header field (Section 5.1), in a form appropriate for arithmetic operation; or 0, if not available.

date_value The term "date_value" denotes the value of the Date header field, in a form appropriate for arithmetic operations. See Section 6.6.1 of [HTTP] for the definition of the Date header field and for requirements regarding responses without it.

now The term "now" means the current value of this implementation's clock (Section 5.6.7 of [HTTP]).

request_time The value of the clock at the time of the request that resulted in the stored response.

response_time The value of the clock at the time the response was received.

A response's age can be calculated in two entirely independent ways:

1. the "apparent_age": response_time minus date_value, if the implementation's clock is reasonably well synchronized to the origin server's clock. If the result is negative, the result is replaced by zero.
2. the "corrected_age_value", if all of the caches along the response path implement HTTP/1.1 or greater. A cache MUST interpret this value relative to the time the request was initiated, not the time that the response was received.

```
apparent_age = max(0, response_time - date_value);

response_delay = response_time - request_time;
corrected_age_value = age_value + response_delay;
```

The corrected_age_value MAY be used as the corrected_initial_age. In circumstances where very old cache implementations that might not correctly insert Age are present, corrected_initial_age can be calculated more conservatively as

```
corrected_initial_age = max(apparent_age, corrected_age_value);
```

The current_age of a stored response can then be calculated by adding the time (in seconds) since the stored response was last validated by the origin server to the corrected_initial_age.

```
resident_time = now - response_time;
current_age = corrected_initial_age + resident_time;
```

4.2.4. Serving Stale Responses

A "stale" response is one that either has explicit expiry information or is allowed to have heuristic expiry calculated, but is not fresh according to the calculations in Section 4.2.

A cache MUST NOT generate a stale response if it is prohibited by an explicit in-protocol directive (e.g., by a no-cache response directive, a must-revalidate response directive, or an applicable s-maxage or proxy-revalidate response directive; see Section 5.2.2).

A cache MUST NOT generate a stale response unless it is disconnected or doing so is explicitly permitted by the client or origin server (e.g., by the max-stale request directive in Section 5.2.1, extension directives such as those defined in [RFC5861], or configuration in accordance with an out-of-band contract).

4.3. Validation

When a cache has one or more stored responses for a requested URI, but cannot serve any of them (e.g., because they are not fresh, or one cannot be chosen; see Section 4.1), it can use the conditional request mechanism (Section 13 of [HTTP]) in the forwarded request to give the next inbound server an opportunity to choose a valid stored response to use, updating the stored metadata in the process, or to replace the stored response(s) with a new response. This process is known as *validating* or *revalidating* the stored response.

4.3.1. Sending a Validation Request

When generating a conditional request for validation, a cache either starts with a request it is attempting to satisfy or — if it is initiating the request independently — synthesizes a request using a stored response by copying the method, target URI, and request header fields identified by the Vary header field (Section 4.1).

It then updates that request with one or more precondition header fields. These contain validator metadata sourced from a stored response(s) that has the same URI. Typically, this will include only the stored response(s) that has the same cache key, although a cache is allowed to validate a response that it cannot choose with the request header fields it is sending (see Section 4.1).

The precondition header fields are then compared by recipients to determine whether any stored response is equivalent to a current representation of the resource.

One such validator is the timestamp given in a Last-Modified header field (Section 8.8.2 of [HTTP]), which can be used in an If-Modified-Since header field for response validation, or in an If-Unmodified-Since or If-Range header field for representation selection (i.e., the client is referring specifically to a previously obtained representation with that timestamp).

Another validator is the entity tag given in an ETag field (Section 8.8.3 of [HTTP]). One or more entity tags, indicating one or more stored responses, can be used in an If-None-Match header field for response validation, or in an If-Match or If-Range header field for representation selection (i.e., the client is referring specifically to one or more previously obtained representations with the listed entity tags).

When generating a conditional request for validation, a cache:

- MUST send the relevant entity tags (using If-Match, If-None-Match, or If-Range) if the entity tags were provided in the stored response(s) being validated.
- SHOULD send the Last-Modified value (using If-Modified-Since) if the request is not for a subrange, a single stored response is being validated, and that response contains a Last-Modified value.
- MAY send the Last-Modified value (using If-Unmodified-Since or If-Range) if the request is for a subrange, a single stored response is being validated, and that response contains only a Last-Modified value (not an entity tag).

In most cases, both validators are generated in cache validation requests, even when entity tags are clearly superior, to allow old intermediaries that do not understand entity tag preconditions to respond appropriately.

4.3.2. Handling a Received Validation Request

Each client in the request chain may have its own cache, so it is common for a cache at an intermediary to receive conditional requests from other (outbound) caches. Likewise, some user agents make use of conditional requests to limit data transfers to recently modified representations or to complete the transfer of a partially retrieved representation.

If a cache receives a request that can be satisfied by reusing a stored 200 (OK) or 206 (Partial Content) response, as per Section 4, the cache SHOULD evaluate any applicable conditional header field preconditions received in that request with respect to the corresponding validators contained within the stored response.

A cache MUST NOT evaluate conditional header fields that only apply to an origin server, occur in a request with semantics that cannot be satisfied with a cached response, or occur in a request with a target resource for which it has no stored responses; such preconditions are likely intended for some other (inbound) server.

The proper evaluation of conditional requests by a cache depends on the received precondition header fields and their precedence. In summary, the If-Match and If-Unmodified-Since conditional header fields are not applicable to a cache, and If-None-Match takes precedence over If-Modified-Since. See Section 13.2.2 of [HTTP] for a complete specification of precondition precedence.

A request containing an If-None-Match header field (Section 13.1.2 of [HTTP]) indicates that the client wants to validate one or more of its own stored responses in comparison to the stored response chosen by the cache (as per Section 4).

If an If-None-Match header field is not present, a request containing an If-Modified-Since header field (Section 13.1.3 of [HTTP]) indicates that the client wants to validate one or more of its own stored responses by modification date.

If a request contains an If-Modified-Since header field and the Last-Modified header field is not present in a stored response, a cache SHOULD use the stored response's Date field value (or, if no Date field is present, the time that the stored response was received) to evaluate the conditional.

A cache that implements partial responses to range requests, as defined in Section 14.2 of [HTTP], also needs to evaluate a received If-Range header field (Section 13.1.5 of [HTTP]) with respect to the cache's chosen response.

When a cache decides to forward a request to revalidate its own stored responses for a request that contains an If-None-Match list of entity tags, the cache MAY combine the received list with a list of entity tags from its own stored set of responses (fresh or stale) and send the union of the two lists as a replacement If-None-Match header field value in the forwarded request. If a stored response contains only partial content, the cache MUST NOT include its entity tag in the union unless the request is for a range that would be fully satisfied by that partial stored response. If the response to the forwarded request is 304 (Not Modified) and has an ETag field value with an entity tag that is not in the client's list, the cache MUST generate a 200 (OK) response for the client by reusing its corresponding stored response, as updated by the 304 response metadata (Section 4.3.4).

4.3.3. Handling a Validation Response

Cache handling of a response to a conditional request depends upon its status code:

- A 304 (Not Modified) response status code indicates that the stored response can be updated and reused; see Section 4.3.4.
- A full response (i.e., one containing content) indicates that none of the stored responses nominated in the conditional request are suitable. Instead, the cache MUST use the full response to satisfy the request. The cache MAY store such a full response, subject to its constraints (see Section 3).
- However, if a cache receives a 5xx (Server Error) response while attempting to validate a response, it can either forward this response to the requesting client or act as if the server failed to respond. In the latter case, the cache can send a previously stored response, subject to its constraints on doing so (see Section 4.2.4), or retry the validation request.

4.3.4. Freshening Stored Responses upon Validation

When a cache receives a 304 (Not Modified) response, it needs to identify stored responses that are suitable for updating with the new information provided, and then do so.

The initial set of stored responses to update are those that could have been chosen for that request — i.e., those that meet the requirements in Section 4, except the last requirement to be fresh, able to be served stale, or just validated.

Then, that initial set of stored responses is further filtered by the first match of:

- If the new response contains one or more *strong validators* (see Section 8.8.1 of [HTTP]), then each of those strong validators identifies a selected representation for update. All the stored responses in the initial set with one of those same strong validators are identified for update. If none of the initial set contains at

least one of the same strong validators, then the cache MUST NOT use the new response to update any stored responses.

- If the new response contains no strong validators but does contain one or more *weak validators*, and those validators correspond to one of the initial set's stored responses, then the most recent of those matching stored responses is identified for update.
- If the new response does not include any form of validator (such as where a client generates an If-Modified-Since request from a source other than the Last-Modified response header field), and there is only one stored response in the initial set, and that stored response also lacks a validator, then that stored response is identified for update.

For each stored response identified, the cache MUST update its header fields with the header fields provided in the 304 (Not Modified) response, as per Section 3.2.

4.3.5. Freshening Responses with HEAD

A response to the HEAD method is identical to what an equivalent request made with a GET would have been, without sending the content. This property of HEAD responses can be used to invalidate or update a cached GET response if the more efficient conditional GET request mechanism is not available (due to no validators being present in the stored response) or if transmission of the content is not desired even if it has changed.

When a cache makes an inbound HEAD request for a target URI and receives a 200 (OK) response, the cache SHOULD update or invalidate each of its stored GET responses that could have been chosen for that request (see Section 4.1).

For each of the stored responses that could have been chosen, if the stored response and HEAD response have matching values for any received validator fields (ETag and Last-Modified) and, if the HEAD response has a Content-Length header field, the value of Content-Length matches that of the stored response, the cache SHOULD update the stored response as described below; otherwise, the cache SHOULD consider the stored response to be stale.

If a cache updates a stored response with the metadata provided in a HEAD response, the cache MUST use the header fields provided in the HEAD response to update the stored response (see Section 3.2).

4.4. Invalidating Stored Responses

Because unsafe request methods (Section 9.2.1 of [HTTP]) such as PUT, POST, or DELETE have the potential for changing state on the origin server, intervening caches are required to invalidate stored responses to keep their contents up to date.

A cache MUST invalidate the target URI (Section 7.1 of [HTTP]) when it receives a non-error status code in response to an unsafe request method (including methods whose safety is unknown).

A cache MAY invalidate other URIs when it receives a non-error status code in response to an unsafe request method (including methods whose safety is unknown). In particular, the URI(s) in the Location and Content-Location response header fields (if present) are candidates for invalidation; other URIs might be discovered through mechanisms not specified in this document. However, a cache MUST NOT trigger an invalidation under these conditions if the origin (Section 4.3.1 of [HTTP]) of the URI to be invalidated differs from that of the target URI (Section 7.1 of [HTTP]). This helps prevent denial-of-service attacks.

Invalidate means that the cache will either remove all stored responses whose target URI matches the given URI or mark them as "invalid" and in need of a mandatory validation before they can be sent in response to a subsequent request.

A "non-error response" is one with a 2xx (Successful) or 3xx (Redirection) status code.

Note that this does not guarantee that all appropriate responses are invalidated globally; a state-changing request would only invalidate responses in the caches it travels through.

5. Field Definitions

This section defines the syntax and semantics of HTTP fields related to caching.

5.1. Age

The "Age" response header field conveys the sender's estimate of the time since the response was generated or successfully validated at the origin server. Age values are calculated as specified in Section 4.2.3.

```
Age = delta-seconds
```

The Age field value is a non-negative integer, representing time in seconds (see Section 1.2.2).

Although it is defined as a singleton header field, a cache encountering a message with a list-based Age field value SHOULD use the first member of the field value, discarding subsequent ones.

If the field value (after discarding additional members, as per above) is invalid (e.g., it contains something other than a non-negative integer), a cache SHOULD ignore the field.

The presence of an Age header field implies that the response was not generated or validated by the origin server for this request. However, lack of an Age header field does not imply the origin was contacted.

5.2. Cache-Control

The "Cache-Control" header field is used to list directives for caches along the request/response chain. Cache directives are unidirectional, in that the presence of a directive in a request does not imply that the same directive is present or copied in the response.

See Section 5.2.3 for information about how Cache-Control directives defined elsewhere are handled.

A proxy, whether or not it implements a cache, MUST pass cache directives through in forwarded messages, regardless of their significance to that application, since the directives might apply to all recipients along the request/response chain. It is not possible to target a directive to a specific cache.

Cache directives are identified by a token, to be compared case-insensitively, and have an optional argument that can use both token and quoted-string syntax. For the directives defined below that define arguments, recipients ought to accept both forms, even if a specific form is required for generation.

```
Cache-Control   = #cache-directive

cache-directive = token [ "=" ( token / quoted-string ) ]
```

For the cache directives defined below, no argument is defined (nor allowed) unless stated otherwise.

5.2.1. Request Directives

This section defines cache request directives. They are advisory; caches MAY implement them, but are not required to.

5.2.1.1. max-age

Argument syntax:

delta-seconds (see Section 1.2.2)

The max-age request directive indicates that the client prefers a response whose age is less than or equal to the specified number of seconds. Unless the max-stale request directive is also present, the client does not wish to receive a stale response.

This directive uses the token form of the argument syntax: e.g., 'max-age=5' not 'max-age="5"'. A sender MUST NOT generate the quoted-string form.

5.2.1.2. max-stale

Argument syntax:

delta-seconds (see Section 1.2.2)

The max-stale request directive indicates that the client will accept a response that has exceeded its freshness lifetime. If a value is present, then the client is willing to accept a response that has exceeded its freshness lifetime by no more than the specified number of seconds. If no value is assigned to max-stale, then the client will accept a stale response of any age.

This directive uses the token form of the argument syntax: e.g., 'max-stale=10' not 'max-stale="10"'. A sender MUST NOT generate the quoted-string form.

5.2.1.3. min-fresh

Argument syntax:

delta-seconds (see Section 1.2.2)

The min-fresh request directive indicates that the client prefers a response whose freshness lifetime is no less than its current age plus the specified time in seconds. That is, the client wants a response that will still be fresh for at least the specified number of seconds.

This directive uses the token form of the argument syntax: e.g., 'min-fresh=20' not 'min-fresh="20"'. A sender MUST NOT generate the quoted-string form.

5.2.1.4. no-cache

The no-cache request directive indicates that the client prefers a stored response not be used to satisfy the request without successful validation on the origin server.

5.2.1.5. no-store

The no-store request directive indicates that a cache MUST NOT store any part of either this request or any response to it. This directive applies to both private and shared caches. "MUST NOT store" in this context means that the cache MUST NOT intentionally store the information in non-volatile storage and MUST make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

This directive is not a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches might not recognize or obey this directive, and communications networks might be vulnerable to eavesdropping.

Note that if a request containing this directive is satisfied from a cache, the no-store request directive does not apply to the already stored response.

5.2.1.6. no-transform

The no-transform request directive indicates that the client is asking for intermediaries to avoid transforming the content, as defined in Section 7.7 of [HTTP].

5.2.1.7. only-if-cached

The only-if-cached request directive indicates that the client only wishes to obtain a stored response. Caches that honor this request directive SHOULD, upon receiving it, respond with either a stored response consistent with the other constraints of the request or a 504 (Gateway Timeout) status code.

5.2.2. Response Directives

This section defines cache response directives. A cache MUST obey the Cache-Control directives defined in this section.

5.2.2.1. max-age

Argument syntax:

delta-seconds (see Section 1.2.2)

The max-age response directive indicates that the response is to be considered stale after its age is greater than the specified number of seconds.

This directive uses the token form of the argument syntax: e.g., 'max-age=5' not 'max-age="5"'. A sender MUST NOT generate the quoted-string form.

5.2.2.2. must-revalidate

The must-revalidate response directive indicates that once the response has become stale, a cache MUST NOT reuse that response to satisfy another request until it has been successfully validated by the origin, as defined by Section 4.3.

The must-revalidate directive is necessary to support reliable operation for certain protocol features. In all circumstances, a cache MUST NOT ignore the must-revalidate directive; in particular, if a cache is disconnected, the cache MUST generate an error response rather than reuse the stale response. The generated status code SHOULD be 504 (Gateway Timeout) unless another error status code is more applicable.

The must-revalidate directive ought to be used by servers if and only if failure to validate a request could cause incorrect operation, such as a silently unexecuted financial transaction.

The must-revalidate directive also permits a shared cache to reuse a response to a request containing an Authorization header field (Section 11.6.2 of [HTTP]), subject to the above requirement on revalidation (Section 3.5).

5.2.2.3. must-understand

The must-understand response directive limits caching of the response to a cache that understands and conforms to the requirements for that response's status code.

A response that contains the must-understand directive SHOULD also contain the no-store directive. When a cache that implements the must-understand directive receives a response that includes it, the cache SHOULD ignore the no-store directive if it understands and implements the status code's caching requirements.

5.2.2.4. no-cache

Argument syntax:

#field-name

The no-cache response directive, in its unqualified form (without an argument), indicates that the response MUST NOT be used to satisfy any other request without forwarding it for validation and receiving a successful response; see Section 4.3.

This allows an origin server to prevent a cache from using the response to satisfy a request without contacting it, even by caches that have been configured to send stale responses.

The qualified form of the no-cache response directive, with an argument that lists one or more field names, indicates that a cache MAY use the response to satisfy a subsequent request, subject to any other restrictions on caching, if the listed header fields are excluded from the subsequent response or the subsequent response has been successfully revalidated with the origin server (updating or removing those fields). This allows an origin

server to prevent the reuse of certain header fields in a response, while still allowing caching of the rest of the response.

The field names given are not limited to the set of header fields defined by this specification. Field names are case-insensitive.

This directive uses the quoted-string form of the argument syntax. A sender SHOULD NOT generate the token form (even if quoting appears not to be needed for single-entry lists).

> **Note:** The qualified form of the directive is often handled by caches as if an unqualified no-cache directive was received; that is, the special handling for the qualified form is not widely implemented.

5.2.2.5. no-store

The no-store response directive indicates that a cache MUST NOT store any part of either the immediate request or the response and MUST NOT use the response to satisfy any other request.

This directive applies to both private and shared caches. "MUST NOT store" in this context means that the cache MUST NOT intentionally store the information in non-volatile storage and MUST make a best-effort attempt to remove the information from volatile storage as promptly as possible after forwarding it.

This directive is not a reliable or sufficient mechanism for ensuring privacy. In particular, malicious or compromised caches might not recognize or obey this directive, and communications networks might be vulnerable to eavesdropping.

Note that the must-understand cache directive overrides no-store in certain circumstances; see Section 5.2.2.3.

5.2.2.6. no-transform

The no-transform response directive indicates that an intermediary (regardless of whether it implements a cache) MUST NOT transform the content, as defined in Section 7.7 of [HTTP].

5.2.2.7. private

Argument syntax:

```
#field-name
```

The unqualified private response directive indicates that a shared cache MUST NOT store the response (i.e., the response is intended for a single user). It also indicates that a private cache MAY store the response, subject to the constraints defined in Section 3, even if the response would not otherwise be heuristically cacheable by a private cache.

If a qualified private response directive is present, with an argument that lists one or more field names, then only the listed header fields are limited to a single user: a shared cache MUST NOT store the listed header fields if they are present in the original response but MAY store the remainder of the response message without those header fields, subject the constraints defined in Section 3.

The field names given are not limited to the set of header fields defined by this specification. Field names are case-insensitive.

This directive uses the quoted-string form of the argument syntax. A sender SHOULD NOT generate the token form (even if quoting appears not to be needed for single-entry lists).

> **Note:** This usage of the word "private" only controls where the response can be stored; it cannot ensure the privacy of the message content. Also, the qualified form of the directive is often handled by caches as if an unqualified private directive was received; that is, the special handling for the qualified form is not widely implemented.

5.2.2.8. proxy-revalidate

The proxy-revalidate response directive indicates that once the response has become stale, a shared cache MUST NOT reuse that response to satisfy another request until it has been successfully validated by the origin, as defined by Section 4.3. This is analogous to must-revalidate (Section 5.2.2.2), except that proxy-revalidate does not apply to private caches.

Note that proxy-revalidate on its own does not imply that a response is cacheable. For example, it might be combined with the public directive (Section 5.2.2.9), allowing the response to be cached while requiring only a shared cache to revalidate when stale.

5.2.2.9. public

The public response directive indicates that a cache MAY store the response even if it would otherwise be prohibited, subject to the constraints defined in Section 3. In other words, public explicitly marks the response as cacheable. For example, public permits a shared cache to reuse a response to a request containing an Authorization header field (Section 3.5).

Note that it is unnecessary to add the public directive to a response that is already cacheable according to Section 3.

If a response with the public directive has no explicit freshness information, it is heuristically cacheable (Section 4.2.2).

5.2.2.10. s-maxage

Argument syntax:

> delta-seconds (see Section 1.2.2)

The s-maxage response directive indicates that, for a shared cache, the maximum age specified by this directive overrides the maximum age specified by either the max-age directive or the Expires header field.

The s-maxage directive incorporates the semantics of the proxy-revalidate response directive (Section 5.2.2.8) for a shared cache. A shared cache MUST NOT reuse a stale response with s-maxage to satisfy another request until it has been successfully validated by the origin, as defined by Section 4.3. This directive also permits a shared cache to reuse a response to a request containing an Authorization header field, subject to the above requirements on maximum age and revalidation (Section 3.5).

This directive uses the token form of the argument syntax: e.g., 's-maxage=10' not 's-maxage="10"'. A sender MUST NOT generate the quoted-string form.

5.2.3. Extension Directives

The Cache-Control header field can be extended through the use of one or more extension cache directives. A cache MUST ignore unrecognized cache directives.

Informational extensions (those that do not require a change in cache behavior) can be added without changing the semantics of other directives.

Behavioral extensions are designed to work by acting as modifiers to the existing base of cache directives. Both the new directive and the old directive are supplied, such that applications that do not understand the new directive will default to the behavior specified by the old directive, and those that understand the new directive will recognize it as modifying the requirements associated with the old directive. In this way, extensions to the existing cache directives can be made without breaking deployed caches.

For example, consider a hypothetical new response directive called "community" that acts as a modifier to the private directive: in addition to private caches, only a cache that is shared by members of the named community is allowed to cache the response. An origin server wishing to allow the UCI community to use an otherwise private response in their shared cache(s) could do so by including

```
Cache-Control: private, community="UCI"
```

A cache that recognizes such a community cache directive could broaden its behavior in accordance with that extension. A cache that does not recognize the community cache directive would ignore it and adhere to the private directive.

New extension directives ought to consider defining:

- What it means for a directive to be specified multiple times,
- When the directive does not take an argument, what it means when an argument is present,
- When the directive requires an argument, what it means when it is missing, and
- Whether the directive is specific to requests, specific to responses, or able to be used in either.

5.2.4. Cache Directive Registry

The "Hypertext Transfer Protocol (HTTP) Cache Directive Registry" defines the namespace for the cache directives. It has been created and is now maintained at https://www.iana.org/assignments/http-cache-directives.

A registration MUST include the following fields:

- Cache Directive Name
- Pointer to specification text

Values to be added to this namespace require IETF Review (see [RFC8126], Section 4.8).

5.3. Expires

The "Expires" response header field gives the date/time after which the response is considered stale. See Section 4.2 for further discussion of the freshness model.

The presence of an Expires header field does not imply that the original resource will change or cease to exist at, before, or after that time.

The Expires field value is an HTTP-date timestamp, as defined in Section 5.6.7 of [HTTP]. See also Section 4.2 for parsing requirements specific to caches.

```
Expires = HTTP-date
```

For example

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

A cache recipient MUST interpret invalid date formats, especially the value "0", as representing a time in the past (i.e., "already expired").

If a response includes a Cache-Control header field with the max-age directive (Section 5.2.2.1), a recipient MUST ignore the Expires header field. Likewise, if a response includes the s-maxage directive (Section 5.2.2.10), a shared cache recipient MUST ignore the Expires header field. In both these cases, the value in Expires is only intended for recipients that have not yet implemented the Cache-Control header field.

An origin server without a clock (Section 5.6.7 of [HTTP]) MUST NOT generate an Expires header field unless its value represents a fixed time in the past (always expired) or its value has been associated with the resource by a system with a clock.

Historically, HTTP required the Expires field value to be no more than a year in the future. While longer freshness lifetimes are no longer prohibited, extremely large values have been demonstrated to cause problems (e.g., clock overflows due to use of 32-bit integers for time values), and many caches will evict a response far sooner than that.

5.4. Pragma

The "Pragma" request header field was defined for HTTP/1.0 caches, so that clients could specify a "no-cache" request (as Cache-Control was not defined until HTTP/1.1).

However, support for Cache-Control is now widespread. As a result, this specification deprecates Pragma.

> **Note:** Because the meaning of "Pragma: no-cache" in responses was never specified, it does not provide a reliable replacement for "Cache-Control: no-cache" in them.

5.5. Warning

The "Warning" header field was used to carry additional information about the status or transformation of a message that might not be reflected in the status code. This specification obsoletes it, as it is not widely generated or surfaced to users. The information it carried can be gleaned from examining other header fields, such as Age.

6. Relationship to Applications and Other Caches

Applications using HTTP often specify additional forms of caching. For example, Web browsers often have history mechanisms such as "Back" buttons that can be used to redisplay a representation retrieved earlier in a session.

Likewise, some Web browsers implement caching of images and other assets within a page view; they may or may not honor HTTP caching semantics.

The requirements in this specification do not necessarily apply to how applications use data after it is retrieved from an HTTP cache. For example, a history mechanism can display a previous representation even if it has expired, and an application can use cached data in other ways beyond its freshness lifetime.

This specification does not prohibit the application from taking HTTP caching into account; for example, a history mechanism might tell the user that a view is stale, or it might honor cache directives (e.g., Cache-Control: no-store).

However, when an application caches data and does not make this apparent to or easily controllable by the user, it is strongly encouraged to define its operation with respect to HTTP cache directives so as not to surprise authors who expect caching semantics to be honored. For example, while it might be reasonable to define an application cache "above" HTTP that allows a response containing Cache-Control: no-store to be reused for requests that are directly related to the request that fetched it (such as those created during the same page load), it would likely be surprising and confusing to users and authors if it were allowed to be reused for requests unrelated in any way to the one from which it was obtained.

7. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns specific to HTTP caching. More general security considerations are addressed in "HTTP/1.1" (Section 11 of [HTTP/1.1]) and "HTTP Semantics" (Section 17 of [HTTP]).

Caches expose an additional attack surface because the contents of the cache represent an attractive target for malicious exploitation. Since cache contents persist after an HTTP request is complete, an attack on the cache can reveal information long after a user believes that the information has been removed from the network. Therefore, cache contents need to be protected as sensitive information.

In particular, because private caches are restricted to a single user, they can be used to reconstruct a user's activity. As a result, it is important for user agents to allow end users to control them, for example, by allowing stored responses to be removed for some or all origin servers.

7.1. Cache Poisoning

Storing malicious content in a cache can extend the reach of an attacker to affect multiple users. Such "cache poisoning" attacks happen when an attacker uses implementation flaws, elevated privileges, or other techniques to insert a response into a cache. This is especially effective when shared caches are used to distribute malicious content to many clients.

One common attack vector for cache poisoning is to exploit differences in message parsing on proxies and in user agents; see Section 6.3 of [HTTP/1.1] for the relevant requirements regarding HTTP/1.1.

7.2. Timing Attacks

Because one of the primary uses of a cache is to optimize performance, its use can "leak" information about which resources have been previously requested.

For example, if a user visits a site and their browser caches some of its responses and then navigates to a second site, that site can attempt to load responses it knows exist on the first site. If they load quickly, it can be assumed that the user has visited that site, or even a specific page on it.

Such "timing attacks" can be mitigated by adding more information to the cache key, such as the identity of the referring site (to prevent the attack described above). This is sometimes called "double keying".

7.3. Caching of Sensitive Information

Implementation and deployment flaws (often led to by the misunderstanding of cache operation) might lead to the caching of sensitive information (e.g., authentication credentials) that is thought to be private, exposing it to unauthorized parties.

Note that the Set-Cookie response header field [COOKIE] does not inhibit caching; a cacheable response with a Set-Cookie header field can be (and often is) used to satisfy subsequent requests to caches. Servers that wish to control caching of these responses are encouraged to emit appropriate Cache-Control response header fields.

8. IANA Considerations

The change controller for the following registrations is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

8.1. Field Name Registration

IANA has updated the "Hypertext Transfer Protocol (HTTP) Field Name Registry" at https://www.iana.org/assignments/http-fields, as described in Section 18.4 of [HTTP], with the field names listed in the table below:

Field Name	Status	Section	Comments
Age	permanent	5.1	
Cache-Control	permanent	5.2	
Expires	permanent	5.3	
Pragma	deprecated	5.4	
Warning	obsoleted	5.5	

Table 1

8.2. Cache Directive Registration

IANA has updated the "Hypertext Transfer Protocol (HTTP) Cache Directive Registry" at https://www.iana.org/assignments/http-cache-directives with the registration procedure per Section 5.2.4 and the cache directive names summarized in the table below.

Cache Directive	Section
max-age	5.2.1.1, 5.2.2.1
max-stale	5.2.1.2
min-fresh	5.2.1.3
must-revalidate	5.2.2.2
must-understand	5.2.2.3
no-cache	5.2.1.4, 5.2.2.4
no-store	5.2.1.5, 5.2.2.5
no-transform	5.2.1.6, 5.2.2.6
only-if-cached	5.2.1.7
private	5.2.2.7
proxy-revalidate	5.2.2.8
public	5.2.2.9
s-maxage	5.2.2.10

Table 2

8.3. Warn Code Registry

IANA has added the following note to the "Hypertext Transfer Protocol (HTTP) Warn Codes" registry at https://www.iana.org/assignments/http-warn-codes stating that "Warning" has been obsoleted:

> *The Warning header field (and the warn codes that it uses) has been obsoleted for HTTP per [RFC9111].*

9. References

9.1. Normative References

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", RFC 9110, DOI 10.17487/RFC9110, June 2022.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/rfc>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc>.

9.2. Informative References

[COOKIE] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc>.

[HTTP/1.1] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", RFC 9112, DOI 10.17487/RFC9112, June 2022.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <https://www.rfc-editor.org/info/rfc>.

[RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale Content", RFC 5861, DOI 10.17487/RFC5861, May 2010, <https://www.rfc-editor.org/info/rfc>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc>.

Appendix A. Collected ABNF

In the collected ABNF below, list rules are expanded per Section 5.6.1 of [HTTP].

```
Age = delta-seconds

Cache-Control = [ cache-directive *( OWS "," OWS cache-directive ) ]

Expires = HTTP-date

HTTP-date = <HTTP-date, see [HTTP], Section 5.6.7>

OWS = <OWS, see [HTTP], Section 5.6.3>

cache-directive = token [ "=" ( token / quoted-string ) ]

delta-seconds = 1*DIGIT

field-name = <field-name, see [HTTP], Section 5.1>

quoted-string = <quoted-string, see [HTTP], Section 5.6.4>

token = <token, see [HTTP], Section 5.6.2>
```

Appendix B. Changes from RFC 7234

Handling of duplicate and conflicting cache directives has been clarified. (Section 4.2.1)

Cache invalidation of the URIs in the Location and Content-Location header fields is no longer required but is still allowed. (Section 4.4)

Cache invalidation of the URIs in the Location and Content-Location header fields is disallowed when the origin is different; previously, it was the host. (Section 4.4)

Handling invalid and multiple Age header field values has been clarified. (Section 5.1)

Some cache directives defined by this specification now have stronger prohibitions against generating the quoted form of their values, since this has been found to create interoperability problems. Consumers of extension cache directives are no longer required to accept both token and quoted-string forms, but they still need to parse them properly for unknown extensions. (Section 5.2)

The public and private cache directives were clarified, so that they do not make responses reusable under any condition. (Section 5.2.2)

The must-understand cache directive was introduced; caches are no longer required to understand the semantics of new response status codes unless it is present. (Section 5.2.2.3)

The Warning response header was obsoleted. Much of the information supported by Warning could be gleaned by examining the response, and the remaining information — although potentially useful — was entirely advisory. In practice, Warning was not added by caches or intermediaries. (Section 5.5)

Acknowledgements

See Appendix "Acknowledgements" of [HTTP], which applies to this document as well.

Index

Authors' Addresses

Roy T. Fielding (editor)
Adobe
345 Park Ave
San Jose, CA 95110
United States of America
Email: fielding@gbiv.com
URI: https://roy.gbiv.com/

Mark Nottingham (editor)
Fastly
Prahran
Australia
Email: mnot@mnot.net
URI: https://www.mnot.net/

Julian Reschke (editor)
greenbytes GmbH
Hafenweg 16
48155 Münster
Germany
Email: julian.reschke@greenbytes.de
URI: https://greenbytes.de/tech/webdav/