Internet Engineering Task Force (IETF)
Request for Comments: 7232
Obsoletes: 2616
Category: Standards Track
ISSN: 2070-1721

R. Fielding, Editor
Adobe
J. Reschke, Editor
greenbytes

June 2014

Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests

Abstract

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. This document defines HTTP/1.1 conditional requests, including metadata header fields for indicating state changes, request header fields for making preconditions on such state, and rules for constructing the responses to a conditional request when one or more preconditions evaluate to false.

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 5741[1].

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7232[2].

Copyright Notice

[1] https://www.rfc-editor.org/rfc/rfc5741.html#section-2
[2] http://www.rfc-editor.org/info/rfc7232
[3] http://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

Conditional requests are HTTP requests [RFC7231] that include one or more header fields indicating a precondition to be tested before applying the method semantics to the target resource. This document defines the HTTP/1.1 conditional request mechanisms in terms of the architecture, syntax notation, and conformance criteria defined in [RFC7230].

Conditional GET requests are the most efficient mechanism for HTTP cache updates [RFC7234]. Conditionals can also be applied to state-changing methods, such as PUT and DELETE, to prevent the "lost update" problem: one client accidentally overwriting the work of another client that has been acting in parallel.

Conditional request preconditions are based on the state of the target resource as a whole (its current value set) or the state as observed in a previously obtained representation (one value in that set). A resource might have multiple current representations, each with its own observable state. The conditional request mechanisms assume that the mapping of requests to a "selected representation" (Section 3 of [RFC7231]) will be consistent over time if the server intends to take advantage of conditionals. Regardless, if the mapping is inconsistent and the server is unable to select the appropriate representation, then no harm will result when the precondition evaluates to false.

The conditional request preconditions defined by this specification (Section 3) are evaluated when applicable to the recipient (Section 5) according to their order of precedence (Section 6).

1.1. Conformance and Error Handling

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Conformance criteria and considerations regarding error handling are defined in Section 2.5 of [RFC7230].

1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [RFC7230], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Appendix B describes rules imported from other documents. Appendix C shows the collected grammar with all list operators expanded to standard ABNF notation.

2. Validators

This specification defines two forms of metadata that are commonly used to observe resource state and test for preconditions: modification dates (Section 2.2) and opaque entity tags (Section 2.3). Additional metadata that reflects resource state has been defined by various extensions of HTTP, such as Web Distributed Authoring and Versioning (WebDAV, [RFC4918]), that are beyond the scope of this specification. A resource metadata value is referred to as a "*validator*" when it is used within a precondition.

2.1. Weak versus Strong

Validators come in two flavors: strong or weak. Weak validators are easy to generate but are far less useful for comparisons. Strong validators are ideal for comparisons but can be very difficult (and occasionally impossible) to generate efficiently. Rather than impose that all forms of resource adhere to the same strength of validator, HTTP exposes the type of validator in use and imposes restrictions on when weak validators can be used as preconditions.

A "strong validator" is representation metadata that changes value whenever a change occurs to the representation data that would be observable in the payload body of a 200 (OK) response to GET.

A strong validator might change for reasons other than a change to the representation data, such as when a semantically significant part of the representation metadata is changed (e.g., Content-Type), but it is in the best interests of the origin server to only change the value when it is necessary to invalidate the stored responses held by remote caches and authoring tools.

Cache entries might persist for arbitrarily long periods, regardless of expiration times. Thus, a cache might attempt to validate an entry using a validator that it obtained in the distant past. A strong validator is unique across all versions of all representations associated with a particular resource over time. However, there is no implication of uniqueness across representations of different resources (i.e., the same strong validator might be in use for representations of multiple resources at the same time and does not imply that those representations are equivalent).

There are a variety of strong validators used in practice. The best are based on strict revision control, wherein each change to a representation always results in a unique node name and revision identifier being assigned before the representation is made accessible to GET. A collision-resistant hash function applied to the representation data is also sufficient if the data is available prior to the response header fields being sent and the digest does not need to be recalculated every time a validation request is received. However, if a resource has distinct representations that differ only in their metadata, such as might occur with content negotiation over media types that happen to share the same data format, then the origin server needs to incorporate additional information in the validator to distinguish those representations.

In contrast, a "weak validator" is representation metadata that might not change for every change to the representation data. This weakness might be due to limitations in how the value is calculated, such as clock resolution, an inability to ensure uniqueness for all possible representations of the resource, or a desire of the resource owner to group representations by some self-determined set of equivalency rather than unique sequences of data. An origin server SHOULD change a weak entity-tag whenever it considers prior representations to be unacceptable as a substitute for the current representation. In other words, a weak entity-tag ought to change whenever the origin server wants caches to invalidate old responses.

For example, the representation of a weather report that changes in content every second, based on dynamic measurements, might be grouped into sets of equivalent representations (from the origin server's perspective) with the same weak validator in order to allow cached representations to be valid for a reasonable period of time (perhaps adjusted dynamically based on server load or weather quality). Likewise, a representation's modification time, if defined with only one-second resolution, might be a weak validator if it is possible for the representation to be modified twice during a single second and retrieved between those modifications.

Likewise, a validator is weak if it is shared by two or more representations of a given resource at the same time, unless those representations have identical representation data. For example, if the origin server sends the

same validator for a representation with a gzip content coding applied as it does for a representation with no content coding, then that validator is weak. However, two simultaneous representations might share the same strong validator if they differ only in the representation metadata, such as when two different media types are available for the same representation data.

Strong validators are usable for all conditional requests, including cache validation, partial content ranges, and "lost update" avoidance. Weak validators are only usable when the client does not require exact equality with previously obtained representation data, such as when validating a cache entry or limiting a web traversal to recent changes.

2.2. Last-Modified

The "Last-Modified" header field in a response provides a timestamp indicating the date and time at which the origin server believes the selected representation was last modified, as determined at the conclusion of handling the request.

```
Last-Modified = HTTP-date
```

An example of its use is

```
Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
```

2.2.1. Generation

An origin server SHOULD send Last-Modified for any selected representation for which a last modification date can be reasonably and consistently determined, since its use in conditional requests and evaluating cache freshness ([RFC7234]) results in a substantial reduction of HTTP traffic on the Internet and can be a significant factor in improving service scalability and reliability.

A representation is typically the sum of many parts behind the resource interface. The last-modified time would usually be the most recent time that any of those parts were changed. How that value is determined for any given resource is an implementation detail beyond the scope of this specification. What matters to HTTP is how recipients of the Last-Modified header field can use its value to make conditional requests and test the validity of locally cached responses.

An origin server SHOULD obtain the Last-Modified value of the representation as close as possible to the time that it generates the Date field value for its response. This allows a recipient to make an accurate assessment of the representation's modification time, especially if the representation changes near the time that the response is generated.

An origin server with a clock MUST NOT send a Last-Modified date that is later than the server's time of message origination (Date). If the last modification time is derived from implementation-specific metadata that evaluates to some time in the future, according to the origin server's clock, then the origin server MUST replace that value with the message origination date. This prevents a future modification date from having an adverse impact on cache validation.

An origin server without a clock MUST NOT assign Last-Modified values to a response unless these values were associated with the resource by some other system or user with a reliable clock.

2.2.2. Comparison

A Last-Modified time, when used as a validator in a request, is implicitly weak unless it is possible to deduce that it is strong, using the following rules:

- The validator is being compared by an origin server to the actual current validator for the representation and,
- That origin server reliably knows that the associated representation did not change twice during the second covered by the presented validator.

or

- The validator is about to be used by a client in an If-Modified-Since, If-Unmodified-Since, or If-Range header field, because the client has a cache entry for the associated representation, and
- That cache entry includes a Date value, which gives the time when the origin server sent the original response, and
- The presented Last-Modified time is at least 60 seconds before the Date value.

or

- The validator is being compared by an intermediate cache to the validator stored in its cache entry for the representation, and
- That cache entry includes a Date value, which gives the time when the origin server sent the original response, and
- The presented Last-Modified time is at least 60 seconds before the Date value.

This method relies on the fact that if two different responses were sent by the origin server during the same second, but both had the same Last-Modified time, then at least one of those responses would have a Date value equal to its Last-Modified time. The arbitrary 60-second limit guards against the possibility that the Date and Last-Modified values are generated from different clocks or at somewhat different times during the preparation of the response. An implementation MAY use a value larger than 60 seconds, if it is believed that 60 seconds is too short.

2.3. ETag

The "ETag" header field in a response provides the current entity-tag for the selected representation, as determined at the conclusion of handling the request. An entity-tag is an opaque validator for differentiating between multiple representations of the same resource, regardless of whether those multiple representations are due to resource state changes over time, content negotiation resulting in multiple representations being valid at the same time, or both. An entity-tag consists of an opaque quoted string, possibly prefixed by a weakness indicator.

```
ETag       = entity-tag

entity-tag = [ weak ] opaque-tag
weak       = %x57.2F ; "W/", case-sensitive
opaque-tag = DQUOTE *etagc DQUOTE
etagc      = %x21 / %x23-7E / obs-text
           ; VCHAR except double quotes, plus obs-text
```

> **Note:** Previously, opaque-tag was defined to be a quoted-string ([RFC2616], Section 3.11); thus, some recipients might perform backslash unescaping. Servers therefore ought to avoid backslash characters in entity tags.

An entity-tag can be more reliable for validation than a modification date in situations where it is inconvenient to store modification dates, where the one-second resolution of HTTP date values is not sufficient, or where modification dates are not consistently maintained.

Examples:

```
ETag: "xyzzy"
ETag: W/"xyzzy"
ETag: ""
```

An entity-tag can be either a weak or strong validator, with strong being the default. If an origin server provides an entity-tag for a representation and the generation of that entity-tag does not satisfy all of the characteristics

of a strong validator (Section 2.1), then the origin server MUST mark the entity-tag as weak by prefixing its opaque value with "W/" (case-sensitive).

2.3.1. Generation

The principle behind entity-tags is that only the service author knows the implementation of a resource well enough to select the most accurate and efficient validation mechanism for that resource, and that any such mechanism can be mapped to a simple sequence of octets for easy comparison. Since the value is opaque, there is no need for the client to be aware of how each entity-tag is constructed.

For example, a resource that has implementation-specific versioning applied to all changes might use an internal revision number, perhaps combined with a variance identifier for content negotiation, to accurately differentiate between representations. Other implementations might use a collision-resistant hash of representation content, a combination of various file attributes, or a modification timestamp that has sub-second resolution.

An origin server SHOULD send an ETag for any selected representation for which detection of changes can be reasonably and consistently determined, since the entity-tag's use in conditional requests and evaluating cache freshness ([RFC7234]) can result in a substantial reduction of HTTP network traffic and can be a significant factor in improving service scalability and reliability.

2.3.2. Comparison

There are two entity-tag comparison functions, depending on whether or not the comparison context allows the use of weak validators:

- *Strong comparison*: two entity-tags are equivalent if both are not weak and their opaque-tags match character-by-character.
- *Weak comparison*: two entity-tags are equivalent if their opaque-tags match character-by-character, regardless of either or both being tagged as "weak".

The example below shows the results for a set of entity-tag pairs and both the weak and strong comparison function results:

ETag 1	ETag 2	Strong Comparison	Weak Comparison
W/"1"	W/"1"	no match	match
W/"1"	W/"2"	no match	no match
W/"1"	"1"	no match	match
"1"	"1"	match	match

2.3.3. Example: Entity-Tags Varying on Content-Negotiated Resources

Consider a resource that is subject to content negotiation (Section 3.4 of [RFC7231]), and where the representations sent in response to a GET request vary based on the Accept-Encoding request header field (Section 5.3.4 of [RFC7231]):

>> Request:

```
GET /index HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip
```

In this case, the response might or might not use the gzip content coding. If it does not, the response might look like:

>> Response:

```
HTTP/1.1 200 OK
Date: Fri, 26 Mar 2010 00:05:00 GMT
ETag: "123-a"
Content-Length: 70
Vary: Accept-Encoding
Content-Type: text/plain

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

An alternative representation that does use gzip content coding would be:

>> Response:

```
HTTP/1.1 200 OK
Date: Fri, 26 Mar 2010 00:05:00 GMT
ETag: "123-b"
Content-Length: 43
Vary: Accept-Encoding
Content-Type: text/plain
Content-Encoding: gzip

...binary data...
```

Note: Content codings are a property of the representation data, so a strong entity-tag for a content-encoded representation has to be distinct from the entity tag of an unencoded representation to prevent potential conflicts during cache updates and range requests. In contrast, transfer codings (Section 4 of [RFC7230]) apply only during message transfer and do not result in distinct entity-tags.

2.4. When to Use Entity-Tags and Last-Modified Dates

In 200 (OK) responses to GET or HEAD, an origin server:

- SHOULD send an entity-tag validator unless it is not feasible to generate one.
- MAY send a weak entity-tag instead of a strong entity-tag, if performance considerations support the use of weak entity-tags, or if it is unfeasible to send a strong entity-tag.
- SHOULD send a Last-Modified value if it is feasible to send one.

In other words, the preferred behavior for an origin server is to send both a strong entity-tag and a Last-Modified value in successful responses to a retrieval request.

A client:

- MUST send that entity-tag in any cache validation request (using If-Match or If-None-Match) if an entity-tag has been provided by the origin server.
- SHOULD send the Last-Modified value in non-subrange cache validation requests (using If-Modified-Since) if only a Last-Modified value has been provided by the origin server.
- MAY send the Last-Modified value in subrange cache validation requests (using If-Unmodified-Since) if only a Last-Modified value has been provided by an HTTP/1.0 origin server. The user agent SHOULD provide a way to disable this, in case of difficulty.

- SHOULD send both validators in cache validation requests if both an entity-tag and a Last-Modified value have been provided by the origin server. This allows both HTTP/1.0 and HTTP/1.1 caches to respond appropriately.

3. Precondition Header Fields

This section defines the syntax and semantics of HTTP/1.1 header fields for applying preconditions on requests. Section 5 defines when the preconditions are applied. Section 6 defines the order of evaluation when more than one precondition is present.

3.1. If-Match

The "If-Match" header field makes the request method conditional on the recipient origin server either having at least one current representation of the target resource, when the field-value is "*", or having a current representation of the target resource that has an entity-tag matching a member of the list of entity-tags provided in the field-value.

An origin server MUST use the strong comparison function when comparing entity-tags for If-Match (Section 2.3.2), since the client intends this precondition to prevent the method from being applied if there have been any changes to the representation data.

```
If-Match = "*" / 1#entity-tag
```

Examples:

```
If-Match: "xyzzy"
If-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-Match: *
```

If-Match is most often used with state-changing methods (e.g., POST, PUT, DELETE) to prevent accidental overwrites when multiple user agents might be acting in parallel on the same resource (i.e., to prevent the "lost update" problem). It can also be used with safe methods to abort a request if the selected representation does not match one already stored (or partially stored) from a prior request.

An origin server that receives an If-Match header field MUST evaluate the condition prior to performing the method (Section 5). If the field-value is "*", the condition is false if the origin server does not have a current representation for the target resource. If the field-value is a list of entity-tags, the condition is false if none of the listed tags match the entity-tag of the selected representation.

An origin server MUST NOT perform the requested method if a received If-Match condition evaluates to false; instead, the origin server MUST respond with either a) the 412 (Precondition Failed) status code or b) one of the 2xx (Successful) status codes if the origin server has verified that a state change is being requested and the final state is already reflected in the current state of the target resource (i.e., the change requested by the user agent has already succeeded, but the user agent might not be aware of it, perhaps because the prior response was lost or a compatible change was made by some other user agent). In the latter case, the origin server MUST NOT send a validator header field in the response unless it can verify that the request is a duplicate of an immediately prior change made by the same user agent.

The If-Match header field can be ignored by caches and intermediaries because it is not applicable to a stored response.

3.2. If-None-Match

The "If-None-Match" header field makes the request method conditional on a recipient cache or origin server either not having any current representation of the target resource, when the field-value is "*", or having a selected representation with an entity-tag that does not match any of those listed in the field-value.

A recipient MUST use the weak comparison function when comparing entity-tags for If-None-Match (Section 2.3.2), since weak entity-tags can be used for cache validation even if there have been changes to the representation data.

```
If-None-Match = "*" / 1#entity-tag
```

Examples:

```
If-None-Match: "xyzzy"
If-None-Match: W/"xyzzy"
If-None-Match: "xyzzy", "r2d2xxxx", "c3piozzzz"
If-None-Match: W/"xyzzy", W/"r2d2xxxx", W/"c3piozzzz"
If-None-Match: *
```

If-None-Match is primarily used in conditional GET requests to enable efficient updates of cached information with a minimum amount of transaction overhead. When a client desires to update one or more stored responses that have entity-tags, the client SHOULD generate an If-None-Match header field containing a list of those entity-tags when making a GET request; this allows recipient servers to send a 304 (Not Modified) response to indicate when one of those stored responses matches the selected representation.

If-None-Match can also be used with a value of "*" to prevent an unsafe request method (e.g., PUT) from inadvertently modifying an existing representation of the target resource when the client believes that the resource does not have a current representation (Section 4.2.1 of [RFC7231]). This is a variation on the "lost update" problem that might arise if more than one client attempts to create an initial representation for the target resource.

An origin server that receives an If-None-Match header field MUST evaluate the condition prior to performing the method (Section 5). If the field-value is "*", the condition is false if the origin server has a current representation for the target resource. If the field-value is a list of entity-tags, the condition is false if one of the listed tags match the entity-tag of the selected representation.

An origin server MUST NOT perform the requested method if the condition evaluates to false; instead, the origin server MUST respond with either a) the 304 (Not Modified) status code if the request method is GET or HEAD or b) the 412 (Precondition Failed) status code for all other request methods.

Requirements on cache handling of a received If-None-Match header field are defined in Section 4.3.2 of [RFC7234].

3.3. If-Modified-Since

The "If-Modified-Since" header field makes a GET or HEAD request method conditional on the selected representation's modification date being more recent than the date provided in the field-value. Transfer of the selected representation's data is avoided if that data has not changed.

```
If-Modified-Since = HTTP-date
```

An example of the field is:

```
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

A recipient MUST ignore If-Modified-Since if the request contains an If-None-Match header field; the condition in If-None-Match is considered to be a more accurate replacement for the condition in If-Modified-Since, and the two are only combined for the sake of interoperating with older intermediaries that might not implement If-None-Match.

A recipient MUST ignore the If-Modified-Since header field if the received field-value is not a valid HTTP-date, or if the request method is neither GET nor HEAD.

A recipient MUST interpret an If-Modified-Since field-value's timestamp in terms of the origin server's clock.

If-Modified-Since is typically used for two distinct purposes: 1) to allow efficient updates of a cached representation that does not have an entity-tag and 2) to limit the scope of a web traversal to resources that have recently changed.

When used for cache updates, a cache will typically use the value of the cached message's Last-Modified field to generate the field value of If-Modified-Since. This behavior is most interoperable for cases where clocks are poorly synchronized or when the server has chosen to only honor exact timestamp matches (due to a problem with Last-Modified dates that appear to go "back in time" when the origin server's clock is corrected or a representation is restored from an archived backup). However, caches occasionally generate the field value based on other data, such as the Date header field of the cached message or the local clock time that the message was received, particularly when the cached message does not contain a Last-Modified field.

When used for limiting the scope of retrieval to a recent time window, a user agent will generate an If-Modified-Since field value based on either its own local clock or a Date header field received from the server in a prior response. Origin servers that choose an exact timestamp match based on the selected representation's Last-Modified field will not be able to help the user agent limit its data transfers to only those changed during the specified window.

An origin server that receives an If-Modified-Since header field SHOULD evaluate the condition prior to performing the method (Section 5). The origin server SHOULD NOT perform the requested method if the selected representation's last modification date is earlier than or equal to the date provided in the field-value; instead, the origin server SHOULD generate a 304 (Not Modified) response, including only those metadata that are useful for identifying or updating a previously cached response.

Requirements on cache handling of a received If-Modified-Since header field are defined in Section 4.3.2 of [RFC7234].

3.4. If-Unmodified-Since

The "If-Unmodified-Since" header field makes the request method conditional on the selected representation's last modification date being earlier than or equal to the date provided in the field-value. This field accomplishes the same purpose as If-Match for cases where the user agent does not have an entity-tag for the representation.

```
If-Unmodified-Since = HTTP-date
```

An example of the field is:

```
If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

A recipient MUST ignore If-Unmodified-Since if the request contains an If-Match header field; the condition in If-Match is considered to be a more accurate replacement for the condition in If-Unmodified-Since, and the two are only combined for the sake of interoperating with older intermediaries that might not implement If-Match.

A recipient MUST ignore the If-Unmodified-Since header field if the received field-value is not a valid HTTP-date.

A recipient MUST interpret an If-Unmodified-Since field-value's timestamp in terms of the origin server's clock.

If-Unmodified-Since is most often used with state-changing methods (e.g., POST, PUT, DELETE) to prevent accidental overwrites when multiple user agents might be acting in parallel on a resource that does not supply entity-tags with its representations (i.e., to prevent the "lost update" problem). It can also be used with safe methods to abort a request if the selected representation does not match one already stored (or partially stored) from a prior request.

An origin server that receives an If-Unmodified-Since header field MUST evaluate the condition prior to performing the method (Section 5). The origin server MUST NOT perform the requested method if the selected representation's last modification date is more recent than the date provided in the field-value; instead the origin server MUST respond with either a) the 412 (Precondition Failed) status code or b) one of the 2xx (Successful)

status codes if the origin server has verified that a state change is being requested and the final state is already reflected in the current state of the target resource (i.e., the change requested by the user agent has already succeeded, but the user agent might not be aware of that because the prior response message was lost or a compatible change was made by some other user agent). In the latter case, the origin server MUST NOT send a validator header field in the response unless it can verify that the request is a duplicate of an immediately prior change made by the same user agent.

The If-Unmodified-Since header field can be ignored by caches and intermediaries because it is not applicable to a stored response.

3.5. If-Range

The "If-Range" header field provides a special conditional request mechanism that is similar to the If-Match and If-Unmodified-Since header fields but that instructs the recipient to ignore the Range header field if the validator doesn't match, resulting in transfer of the new selected representation instead of a 412 (Precondition Failed) response. If-Range is defined in Section 3.2 of [RFC7233].

4. Status Code Definitions

4.1. 304 Not Modified

The *304 (Not Modified)* status code indicates that a conditional GET or HEAD request has been received and would have resulted in a 200 (OK) response if it were not for the fact that the condition evaluated to false. In other words, there is no need for the server to transfer a representation of the target resource because the request indicates that the client, which made the request conditional, already has a valid representation; the server is therefore redirecting the client to make use of that stored representation as if it were the payload of a 200 (OK) response.

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

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

Requirements on a cache that receives a 304 response are defined in Section 4.3.4 of [RFC7234]. If the conditional request originated with an outbound client, such as a user agent with its own cache sending a conditional GET to a shared proxy, then the proxy SHOULD forward the 304 response to that client.

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

4.2. 412 Precondition Failed

The *412 (Precondition Failed)* status code indicates that one or more conditions given in the request header fields evaluated to false when tested on the server. This response code allows the client to place preconditions on the current resource state (its current representations and metadata) and, thus, prevent the request method from being applied if the target resource is in an unexpected state.

5. Evaluation

Except when excluded below, a recipient cache or origin server MUST evaluate received request preconditions after it has successfully performed its normal request checks and just before it would perform the action associated with the request method. A server MUST ignore all received preconditions if its response to the same request without those conditions would have been a status code other than a 2xx (Successful) or 412 (Precondition Failed). In other words, redirects and failures take precedence over the evaluation of preconditions in conditional requests.

A server that is not the origin server for the target resource and cannot act as a cache for requests on the target resource MUST NOT evaluate the conditional request header fields defined by this specification, and it MUST forward them if the request is forwarded, since the generating client intends that they be evaluated by a server that can provide a current representation. Likewise, a server MUST ignore the conditional request header fields defined by this specification when received with a request method that does not involve the selection or modification of a selected representation, such as CONNECT, OPTIONS, or TRACE.

Conditional request header fields that are defined by extensions to HTTP might place conditions on all recipients, on the state of the target resource in general, or on a group of resources. For instance, the "If" header field in WebDAV can make a request conditional on various aspects of multiple resources, such as locks, if the recipient understands and implements that field ([RFC4918], Section 10.4).

Although conditional request header fields are defined as being usable with the HEAD method (to keep HEAD's semantics consistent with those of GET), there is no point in sending a conditional HEAD because a successful response is around the same size as a 304 (Not Modified) response and more useful than a 412 (Precondition Failed) response.

6. Precedence

When more than one conditional request header field is present in a request, the order in which the fields are evaluated becomes important. In practice, the fields defined in this document are consistently implemented in a single, logical order, since "lost update" preconditions have more strict requirements than cache validation, a validated cache is more efficient than a partial response, and entity tags are presumed to be more accurate than date validators.

A recipient cache or origin server MUST evaluate the request preconditions defined by this specification in the following order:

1. When recipient is the origin server and If-Match is present, evaluate the If-Match precondition:
   - if true, continue to step 3
   - if false, respond 412 (Precondition Failed) unless it can be determined that the state-changing request has already succeeded (see Section 3.1)
2. When recipient is the origin server, If-Match is not present, and If-Unmodified-Since is present, evaluate the If-Unmodified-Since precondition:
   - if true, continue to step 3
   - if false, respond 412 (Precondition Failed) unless it can be determined that the state-changing request has already succeeded (see Section 3.4)
3. When If-None-Match is present, evaluate the If-None-Match precondition:
   - if true, continue to step 5
   - if false for GET/HEAD, respond 304 (Not Modified)
   - if false for other methods, respond 412 (Precondition Failed)
4. When the method is GET or HEAD, If-None-Match is not present, and If-Modified-Since is present, evaluate the If-Modified-Since precondition:
   - if true, continue to step 5
   - if false, respond 304 (Not Modified)
5. When the method is GET and both Range and If-Range are present, evaluate the If-Range precondition:
   - if the validator matches and the Range specification is applicable to the selected representation, respond 206 (Partial Content) [RFC7233]
6. Otherwise,
   - all conditions are met, so perform the requested action and respond according to its success or failure.

Any extension to HTTP/1.1 that defines additional conditional request header fields ought to define its own expectations regarding the order for evaluating such fields in relation to those defined in this document and other conditionals that might be found in practice.

7. IANA Considerations

7.1. Status Code Registration

The "Hypertext Transfer Protocol (HTTP) Status Code Registry" located at <http://www.iana.org/assignments/http-status-codes> has been updated with the registrations below:

Value	Description	Reference
304	Not Modified	Section 4.1
412	Precondition Failed	Section 4.2

7.2. Header Field Registration

HTTP header fields are registered within the "Message Headers" registry maintained at <http://www.iana.org/assignments/message-headers/>.

This document defines the following HTTP header fields, so their associated registry entries have been updated according to the permanent registrations below (see [BCP90]):

Header Field Name	Protocol	Status	Reference
ETag	http	standard	Section 2.3
If-Match	http	standard	Section 3.1
If-Modified-Since	http	standard	Section 3.3
If-None-Match	http	standard	Section 3.2
If-Unmodified-Since	http	standard	Section 3.4
Last-Modified	http	standard	Section 2.2

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

8. Security Considerations

This section is meant to inform developers, information providers, and users of known security concerns specific to the HTTP conditional request mechanisms. More general security considerations are addressed in HTTP "Message Syntax and Routing" [RFC7230] and "Semantics and Content" [RFC7231].

The validators defined by this specification are not intended to ensure the validity of a representation, guard against malicious changes, or detect man-in-the-middle attacks. At best, they enable more efficient cache updates and optimistic concurrent writes when all participants are behaving nicely. At worst, the conditions will fail and the client will receive a response that is no more harmful than an HTTP exchange without conditional requests.

An entity-tag can be abused in ways that create privacy risks. For example, a site might deliberately construct a semantically invalid entity-tag that is unique to the user or user agent, send it in a cacheable response with a long freshness time, and then read that entity-tag in later conditional requests as a means of re-identifying that user or user agent. Such an identifying tag would become a persistent identifier for as long as the user agent retained the original cache entry. User agents that cache representations ought to ensure that the cache is cleared or replaced whenever the user performs privacy-maintaining actions, such as clearing stored cookies or changing to a private browsing mode.

9. Acknowledgments

See Section 10 of [RFC7230].

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, June 2014.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, June 2014.

10.2. Informative References

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

[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, June 1999.

[RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, June 2007.

Appendix A. Changes from RFC 2616

The definition of validator weakness has been expanded and clarified. (Section 2.1)

Weak entity-tags are now allowed in all requests except range requests. (Sections 2.1 and 3.2)

The ETag header field ABNF has been changed to not use quoted-string, thus avoiding escaping issues. (Section 2.3)

ETag is defined to provide an entity tag for the selected representation, thereby clarifying what it applies to in various situations (such as a PUT response). (Section 2.3)

The precedence for evaluation of conditional requests has been defined. (Section 6)

Appendix B. Imported ABNF

The following core rules are included by reference, as defined in Appendix B.1 of [RFC5234]: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTL (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), OCTET (any 8-bit sequence of data), SP (space), and VCHAR (any visible US-ASCII character).

The rules below are defined in [RFC7230]:

```
OWS            = <OWS, see [RFC7230], Section 3.2.3>
obs-text       = <obs-text, see [RFC7230], Section 3.2.6>
```

The rules below are defined in other parts:

```
HTTP-date      = <HTTP-date, see [RFC7231], Section 7.1.1.1>
```

Appendix C. Collected ABNF

In the collected ABNF below, list rules are expanded as per Section 1.2 of [RFC7230].

```
ETag = entity-tag

HTTP-date = <HTTP-date, see [RFC7231], Section 7.1.1.1>

If-Match = "*" / ( *( "," OWS ) entity-tag *( OWS "," [ OWS
 entity-tag ] ) )
If-Modified-Since = HTTP-date
If-None-Match = "*" / ( *( "," OWS ) entity-tag *( OWS "," [ OWS
 entity-tag ] ) )
If-Unmodified-Since = HTTP-date

Last-Modified = HTTP-date

OWS = <OWS, see [RFC7230], Section 3.2.3>

entity-tag = [ weak ] opaque-tag
etagc = "!" / %x23-7E ; '#'-'~'
 / obs-text

obs-text = <obs-text, see [RFC7230], Section 3.2.6>
opaque-tag = DQUOTE *etagc DQUOTE

weak = %x57.2F ; W/
```

Index

Authors' Addresses

Roy T. Fielding (editor)
Adobe Systems Incorporated
345 Park Ave
San Jose, CA 95110
USA
Email: fielding@gbiv.com
URI: http://roy.gbiv.com/

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany
Email: julian.reschke@greenbytes.de
URI: http://greenbytes.de/tech/webdav/