HTTPbis Working Group
Internet-Draft
Obsoletes: 2616 (if approved)
Intended status: Standards Track
Expires: January 17, 2013

R. Fielding, Ed.
Adobe
Y. Lafon, Ed.
W3C
J. Reschke, Ed.
greenbytes

July 16, 2012

# HTTP/1.1, part 4: Conditional Requests
# draft-ietf-httpbis-p4-conditional-20

## Abstract

The Hypertext Transfer Protocol (HTTP) is an 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.

## Editorial Note (To be removed by RFC Editor)

Discussion of this draft takes place on the HTTPBIS working group mailing list (ietf-http-wg@w3.org), which is archived at <http://lists.w3.org/Archives/Public/ietf-http-wg/>.

The current issues list is at <http://tools.ietf.org/wg/httpbis/trac/report/3> and related documents (including fancy diffs) can be found at <http://tools.ietf.org/wg/httpbis/>.

The changes in this draft are summarized in Appendix D.1.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference

material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 17, 2013.

Copyright Notice

Table of Contents

# 1. Introduction

Conditional requests are HTTP requests [Part2] that include one or more header fields indicating a precondition to be tested before applying the method semantics to the target resource. Each precondition is based on metadata that is expected to change if the selected representation of the target resource is changed. This document defines the HTTP/1.1 conditional request mechanisms in terms of the architecture, syntax notation, and conformance criteria defined in [Part1].

Conditional GET requests are the most efficient mechanism for HTTP cache updates [Part6]. 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 corresponding representations 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.

We use the term "selected representation" to refer to the current representation of the target resource that would have been selected in a successful response if the same request had used the method GET and had excluded all of the conditional request header fields. The conditional request preconditions are evaluated by comparing the values provided in the request header fields to the current metadata for the selected representation.

## 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].

This specification targets conformance criteria according to the role of a participant in HTTP communication. Hence, HTTP requirements are placed on senders, recipients, clients, servers, user agents, intermediaries, origin servers, proxies, gateways, or caches, depending on what behavior is being constrained by the requirement. See Section 2 of [Part1] for definitions of these terms.

The verb "generate" is used instead of "send" where a requirement differentiates between creating a protocol element and merely forwarding a received element downstream.

An implementation is considered conformant if it complies with all of the requirements associated with the roles it partakes in HTTP. Note that SHOULD-level requirements are relevant here, unless one of the documented exceptions is applicable.

This document also uses ABNF to define valid protocol elements (Section 1.2). In addition to the prose requirements placed upon them, senders MUST NOT generate protocol elements that do not match the grammar defined by the ABNF rules for those protocol elements that are applicable to the sender's role. If a received protocol element is processed, the recipient MUST be able to parse any value that would match the ABNF rules for that protocol element, excluding only those rules not applicable to the recipient's role.

Unless noted otherwise, a recipient MAY attempt to recover a usable protocol element from an invalid construct. HTTP does not define specific error handling mechanisms except when they have a direct impact on security, since different applications of the protocol require different error handling strategies. For example, a Web browser might wish to transparently recover from a response where the Location header field doesn't parse according to the ABNF, whereas a systems control client might consider any form of error recovery to be dangerous.

### 1.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with the list rule extension defined in Section 1.2 of [Part1]. Appendix B describes rules imported from other documents. Appendix C shows the collected ABNF with the list rule expanded.

## 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 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 a representation metadata value that MUST be changed to a new, previously unused or guaranteed unique, value whenever a change occurs to the representation data such that a change would be observable in the payload body of a 200 (OK) response to GET.

A strong validator MAY be changed for other reasons, 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. A strong validator MUST be unique across all representations of a given resource, such that no two representations of that resource share the same validator unless their payload body would be identical.

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 MUST be 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 SHOULD incorporate additional

information in the validator to distinguish those representations and avoid confusing cache behavior.

In contrast, a "weak validator" is a representation metadata value that might not be changed for every change to the representation data. This weakness might be due to limitations in how the value is calculated, such as clock resolution or an inability to ensure uniqueness for all possible representations of the resource, or due to a desire by 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.

A "use" of a validator occurs when either a client generates a request and includes the validator in a precondition or when a server compares two validators. Weak validators are only usable in contexts that do not depend on exact equality of a representation's payload body. Strong validators are usable and preferred for all conditional requests, including cache validation, partial content ranges, and "lost update" avoidance.

## 2.2. Last-Modified

The "Last-Modified" header field indicates the date and time at which the origin server believes the selected representation was last modified.

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

An example of its use is

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

#### 2.2.1. Generation

Origin servers 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 ([Part6]) 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 header field, because the client has a cache entry, or If-Range 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 provides the current entity-tag for the selected representation. 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 the requirements for a strong validator (Section 2.1), then that entity-tag MUST be marked 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 filesystem attributes, or a modification timestamp that has sub-second

resolution.

Origin servers SHOULD send 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 ([Part6]) 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 the comparison context allows the use of weak validators or not:

- The strong comparison function: in order to be considered equal, both opaque-tags MUST be identical character-by-character, and both MUST NOT be weak.

- The weak comparison function: in order to be considered equal, both opaque-tags MUST be identical character-by-character, but either or both of them MAY be tagged as "weak" without affecting the result.

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 8 of [Part2]), and where the representations returned upon a GET request vary based on the Accept-Encoding request header field (Section 9.3 of [Part2]):

>> 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: Thu, 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: Thu, 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, so therefore an entity-tag of an encoded representation has to be distinct from an unencoded representation to prevent conflicts during cache updates and range requests. In contrast, transfer codings (Section 4 of [Part1]) apply only during message transfer and do not require distinct entity-tags.

## 2.4. Rules for When to Use Entity-tags and Last-Modified Dates

We adopt a set of rules and recommendations for origin servers, clients, and caches regarding when various validator types ought to be used, and for what purposes.

HTTP/1.1 origin servers:

- 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 HTTP/1.1 origin server is to send both a strong entity-tag and a Last-Modified value.

HTTP/1.1 clients:

- MUST use that entity-tag in any cache-conditional request (using If-Match or If-None-Match) if an entity-tag has been provided by the origin server.
- SHOULD use the Last-Modified value in non-subrange cache-conditional requests (using If-Modified-Since) if only a Last-Modified value has been provided by the origin server.
- MAY use the Last-Modified value in subrange cache-conditional 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 use both validators in cache-conditional 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.

An HTTP/1.1 origin server, upon receiving a conditional request that includes both a Last-Modified date (e.g., in an If-Modified-Since or If-Unmodified-Since header field) and one or more entity-tags (e.g., in an If-Match, If-None-Match, or If-Range header field) as cache validators, MUST NOT return a response status code of 304 (Not Modified) unless doing so is consistent with all of the conditional header fields in the request.

An HTTP/1.1 caching proxy, upon receiving a conditional request that includes both a Last-Modified date and one or more entity-tags as cache validators, MUST NOT return a locally cached response to the client unless that cached response is consistent with all of the conditional header fields in the request.

> Note: The general principle behind these rules is that HTTP/1.1 servers and clients ought to transmit as much non-redundant

information as is available in their responses and requests. HTTP/1.1 systems receiving this information will make the most conservative assumptions about the validators they receive.

HTTP/1.0 clients and caches might ignore entity-tags. Generally, last-modified values received or used by these systems will support transparent and efficient caching, and so HTTP/1.1 origin servers still ought to provide Last-Modified values.

## 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 the order of evaluation when more than one precondition is present in a request.

### 3.1. If-Match

The "If-Match" header field can be used to make a request method conditional on the current existence or value of an entity-tag for one or more representations of the target resource.

If-Match is generally useful for resource update requests, such as PUT requests, as a means for protecting against accidental overwrites when multiple clients are acting in parallel on the same resource (i.e., the "lost update" problem). An If-Match field-value of "*" places the precondition on the existence of any current representation for the target resource.

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

The If-Match condition is met if and only if any of the entity-tags listed in the If-Match field value match the entity-tag of the selected representation for the target resource (as per Section 2.3.2), or if "*" is given and any current representation exists for the target resource.

If the condition is met, the server MAY perform the request method as if the If-Match header field was not present.

Origin servers MUST NOT perform the requested method if the condition is not met; instead they MUST respond with the 412 (Precondition Failed) status code.

Proxy servers using a cached response as the selected representation MUST NOT perform the requested method if the condition is not met; instead, they MUST forward the request towards the origin server.

If the request would, without the If-Match header field, result in anything other than a 2xx (Successful) or 412 (Precondition Failed) status code, then the If-Match header field MUST be ignored.

Examples:

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

### 3.2. If-None-Match

The "If-None-Match" header field can be used to make a request method conditional on not matching any of the current entity-tag values for representations of the target resource.

If-None-Match is primarily used in conditional GET requests to enable efficient updates of cached information with a minimum amount of transaction overhead. A client that has one or more representations previously obtained from the target resource can send If-None-Match with a list of the associated entity-tags in the hope of receiving a 304 (Not Modified) response if at least one of those representations 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. 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.

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

The If-None-Match condition is met if and only if none of the entity-tags listed in the If-None-Match field value match the entity-tag of the selected representation for the target resource (as per Section 2.3.2), or if "*" is given and no current representation exists for that resource.

If the condition is not met, the server MUST NOT perform the requested method. Instead, if the request method was GET or HEAD, the server SHOULD respond with a 304 (Not Modified) status code, including the cache-related header fields (particularly ETag) of the selected representation that has a matching entity-tag. For all other request methods, the server MUST respond with a 412 (Precondition Failed) status code.

If the condition is met, the server MAY perform the requested method as if the If-None-Match header field did not exist, but MUST also ignore any If-Modified-Since header field(s) in the request. That is, if no entity-tags match, then the server MUST NOT return a 304 (Not Modified) response.

If the request would, without the If-None-Match header field, result in anything other than a 2xx (Successful) or 304 (Not Modified) status code, then the If-None-Match header field MUST be ignored. (See Section 2.4 for a discussion of server behavior when both If-Modified-Since and If-None-Match appear in the same request.)

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: *
```

### 3.3. If-Modified-Since

The "If-Modified-Since" header field can be used with GET or HEAD to make the method conditional by modification date: if the selected representation has not been modified since the time specified in this field, then do not perform the request method; instead, respond as detailed below.

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

An example of the field is:

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

A GET method with an If-Modified-Since header field and no Range header field requests that the selected representation be transferred only if it has been modified since the date given by the If-Modified-Since header field. The algorithm for determining this includes the following cases:

1. If the request would normally result in anything other than a 200 (OK) status code, or if the passed If-Modified-Since date is invalid, the response is exactly the same as for a normal GET. A date which is later than the server’s current time is invalid.

2. If the selected representation has been modified since the If-Modified-Since date, the response is exactly the same as for a normal GET.

3. If the selected representation has not been modified since a valid If-Modified-Since date, the server SHOULD return a 304 (Not Modified) response.

The purpose of this feature is to allow efficient updates of cached information with a minimum amount of transaction overhead.

Note: The Range header field modifies the meaning of If-Modified-Since; see Section 5.4 of [Part5] for full details.

Note: If-Modified-Since times are interpreted by the server, whose clock might not be synchronized with the client.

Note: When handling an If-Modified-Since header field, some servers will use an exact date comparison function, rather than a less-than function, for deciding whether to send a 304 (Not Modified) response. To get best results when sending an If-Modified-Since header field for cache validation, clients are advised to use the exact date string received in a previous Last-Modified header field whenever possible.

Note: If a client uses an arbitrary date in the If-Modified-Since header field instead of a date taken from the Last-Modified header field for the same request, the client needs to be aware that this date is interpreted in the server's understanding of time. Unsynchronized clocks and rounding problems, due to the different encodings of time between the client and server, are concerns. This includes the possibility of race conditions if the document has changed between the time it was first requested and the If-Modified-Since date of a subsequent request, and the possibility of clock-skew-related problems if the If-Modified-Since date is derived from the client's clock without correction to the server's clock. Corrections for different time bases between client and server are at best approximate due to network latency.

### 3.4. If-Unmodified-Since

The "If-Unmodified-Since" header field can be used to make a request method conditional by modification date: if the selected representation has been modified since the time specified in this field, then the server MUST NOT perform the requested operation and MUST instead respond with the 412 (Precondition Failed) status code. If the selected representation has not been modified since the time specified in this field, the server SHOULD perform the request method as if the If-Unmodified-Since header field were not present.

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

An example of the field is:

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

If a request normally (i.e., in absence of the If-Unmodified-Since header field) would result in anything other than a 2xx (Successful) or 412 (Precondition Failed) status code, the If-Unmodified-Since header field SHOULD be ignored.

If the specified date is invalid, the header field MUST be ignored.

### 3.5. If-Range

The "If-Range" header field provides a special conditional request mechanism that is similar to If-Match and If-Unmodified-Since but specific to HTTP range requests. If-Range is defined in Section 5.3 of [Part5].

## 4. Status Code Definitions

### 4.1. 304 Not Modified

The 304 status code indicates that a conditional GET request has been received and would have resulted in a 200 (OK) response if it were not for the fact that the condition has evaluated to false. In other words, there is no need for the server to transfer a representation of the target resource because the client’s request indicates that it already has a valid representation, as indicated by the 304 response header fields, and is therefore redirecting the client to make use of that stored representation as if it were the payload of a 200 response. The 304 response MUST NOT contain a message-body, and thus is always terminated by the first empty line after the header fields.

A 304 response MUST include a Date header field (Section 9.10 of [Part2]) unless the origin server does not have a clock that can provide a reasonable approximation of the current time. If a 200 (OK) response to the same request would have included any of the header fields Cache-Control, Content-Location, ETag, Expires, or Vary, then those same header fields MUST be sent in a 304 response.

Since the goal of a 304 response is to minimize information transfer when the recipient already has one or more cached representations, the response SHOULD NOT include representation metadata other than the above listed fields unless said metadata exists for the purpose of guiding cache updates (e.g., future HTTP extensions).

If the recipient of a 304 response does not have a cached representation corresponding to the entity-tag indicated by the 304

response, then the recipient MUST NOT use the 304 to update its own cache. If this 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 304 response MAY be forwarded to that client. Otherwise, the recipient MUST disregard the 304 response and repeat the request without any preconditions.

If a cache uses a received 304 response to update a cache entry, the cache MUST update the entry to reflect any new field values given in the response.

### 4.2. 412 Precondition Failed

The 412 status code indicates that one or more preconditions 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. 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, due to the fact that entity tags are presumed to be more accurate than date validators. For example, the only reason to send both If-Modified-Since and If-None-Match in the same GET request is to support intermediary caches that might not have implemented If-None-Match, so it makes sense to ignore the If-Modified-Since when entity tags are understood and available for the selected representation.

The general rule of conditional precedence is that exact match conditions are evaluated before cache-validating conditions and, within that order, last-modified conditions are only evaluated if the corresponding entity tag condition is not present (or not applicable because the selected representation does not have an entity tag).

Specifically, the fields defined by this specification are evaluated as follows:

1. When If-Match is present, evaluate it:

    * if true, continue to step 3

    * if false, respond 412 (Precondition Failed)

2. When If-Match is not present and If-Unmodified-Since is present, evaluate it:

   * if true, continue to step 3

   * if false, respond 412 (Precondition Failed)

3. When the method is GET and both Range and If-Range are present, evaluate it:

   * if the validator matches, respond 206 (Partial Content)

   * if the validator does not match, respond 200 (OK)

4. When If-None-Match is present, evaluate it:

   * if true, all conditions are met

   * if false for GET/HEAD, respond 304 (Not Modified)

   * if false for other methods, respond 412 (Precondition Failed)

5. When the method is GET or HEAD, If-None-Match is not present, and If-Modified-Since is present, evaluate it:

   * if true, all conditions are met

   * if false, respond 304 (Not Modified)

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.

# 6. IANA Considerations

## 6.1. Status Code Registration

The HTTP Status Code Registry located at <http://www.iana.org/assignments/http-status-codes> shall be updated with the registrations below:

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

### 6.2. Header Field Registration

The Message Header Field Registry located at <http://www.iana.org/assignments/message-headers/message-header-index.html> shall be updated with the permanent registrations below (see [RFC3864]):

| 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".

## 7. Security Considerations

No additional security considerations have been identified beyond those applicable to HTTP in general [Part1].

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.

## 8. Acknowledgments

See Section 9 of [Part1].

## 9. References

### 9.1. Normative References

[Part1] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "HTTP/1.1, part 1: Message Routing and Syntax"", draft-ietf-httpbis-p1-messaging-20 (work in progress), July 2012.

[Part2] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "HTTP/1.1, part 2: Semantics and Payloads", draft-ietf-httpbis-p2-semantics-20 (work in progress), July 2012.

[Part5] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "HTTP/1.1, part 5: Range Requests", draft-ietf-httpbis-p5-range-20 (work in progress), July 2012.

[Part6] Fielding, R., Ed., Lafon, Y., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1, part 6: Caching", draft-ietf-httpbis-p6-cache-20 (work in progress), July 2012.

[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.

### 9.2. Informative References

[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.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

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

## Appendix A. Changes from RFC 2616

Allow weak entity-tags in all requests except range requests (Sections 2.1 and 3.2).

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

Change ABNF productions for header fields to only define the field value. (Section 3)

## 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 [Part1]:

```
  OWS           = <OWS, defined in [Part1], Section 3.2.1>
  obs-text      = <obs-text, defined in [Part1], Section 3.2.4>
```

The rules below are defined in other parts:

```
  HTTP-date     = <HTTP-date, defined in [Part2], Section 5.1>
```

## Appendix C. Collected ABNF

```
ETag = entity-tag

HTTP-date = <HTTP-date, defined in [Part2], Section 5.1>

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, defined in [Part1], Section 3.2.1>

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

obs-text = <obs-text, defined in [Part1], Section 3.2.4>
opaque-tag = DQUOTE *etagc DQUOTE

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

## Appendix D. Change Log (to be removed by RFC Editor before publication)

Changes up to the first Working Group Last Call draft are summarized in <http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-19#appendix-C>.

### D.1. Since draft-ietf-httpbis-p4-conditional-19

Closed issues:

- <http://tools.ietf.org/wg/httpbis/trac/ticket/241>: "Need to clarify eval order/interaction of conditional headers"
- <http://tools.ietf.org/wg/httpbis/trac/ticket/354>: "ETags and Conditional Requests"
- <http://tools.ietf.org/wg/httpbis/trac/ticket/361>: "ABNF requirements for recipients"
- <http://tools.ietf.org/wg/httpbis/trac/ticket/363>: "Rare cases"
- <http://tools.ietf.org/wg/httpbis/trac/ticket/365>: "Conditional Request Security Considerations"
- <http://tools.ietf.org/wg/httpbis/trac/ticket/371>: "If-Modified-Since lacks definition for method != GET"
- <http://tools.ietf.org/wg/httpbis/trac/ticket/372>: "refactor conditional header field descriptions"

## 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/

Yves Lafon (editor)
World Wide Web Consortium
W3C / ERCIM
2004, rte des Lucioles
Sophia-Antipolis, AM 06902
France

EMail: ylafon@w3.org
URI: http://www.raubacapeu.net/people/yves/

Julian F. Reschke (editor)
greenbytes GmbH
Hafenweg 16
Muenster, NW 48155
Germany

EMail: julian.reschke@greenbytes.de
URI: http://greenbytes.de/tech/webdav/