

HTTP Working Group  
Internet-Draft  
Intended status: Experimental  
Expires: January 3, 2019

K. Oku  
Fastly  
Y. Weiss  
Akamai  
July 2, 2018

Cache Digests for HTTP/2  
draft-ietf-httpbis-cache-digest-05

Abstract

This specification defines a HTTP/2 frame type to allow clients to inform the server of their cache's contents. Servers can then use this to inform their choices of what to push to clients.

Note to Readers

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

Working Group information can be found at <http://httpwg.github.io/> ; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/cache-digest> .

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 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
1.1. Notational Conventions . . . . .	3
2. The CACHE_DIGEST Frame . . . . .	3
2.1. Client Behavior . . . . .	4
2.1.1. Creating a digest . . . . .	4
2.1.2. Adding a URL to the Digest-Value . . . . .	5
2.1.3. Removing a URL to the Digest-Value . . . . .	7
2.1.4. Computing a fingerprint value . . . . .	8
2.1.5. Computing the key . . . . .	9
2.1.6. Computing a Hash Value . . . . .	9
2.1.7. Computing an Alternative Hash Value . . . . .	9
2.2. Server Behavior . . . . .	10
2.2.1. Querying the Digest for a Value . . . . .	10
3. The SETTINGS_SENDING_CACHE_DIGEST SETTINGS Parameter . . . . .	11
4. The SETTINGS_ACCEPT_CACHE_DIGEST SETTINGS Parameter . . . . .	12
5. IANA Considerations . . . . .	12
6. Security Considerations . . . . .	13
7. References . . . . .	13
7.1. Normative References . . . . .	13
7.2. Informative References . . . . .	14
Appendix A. Encoding the CACHE_DIGEST frame as an HTTP Header . . . . .	15
Appendix B. Changes . . . . .	16
B.1. Since draft-ietf-httpbis-cache-digest-04 . . . . .	16
B.2. Since draft-ietf-httpbis-cache-digest-03 . . . . .	16
B.3. Since draft-ietf-httpbis-cache-digest-02 . . . . .	16
B.4. Since draft-ietf-httpbis-cache-digest-01 . . . . .	16
B.5. Since draft-ietf-httpbis-cache-digest-00 . . . . .	17
Appendix C. Acknowledgements . . . . .	17
Authors' Addresses . . . . .	17

## 1. Introduction

HTTP/2 [RFC7540] allows a server to "push" synthetic request/response pairs into a client's cache optimistically. While there is strong interest in using this facility to improve perceived Web browsing

performance, it is sometimes counterproductive because the client might already have cached the "pushed" response.

When this is the case, the bandwidth used to "push" the response is effectively wasted, and represents opportunity cost, because it could be used by other, more relevant responses. HTTP/2 allows a stream to be cancelled by a client using a RST\_STREAM frame in this situation, but there is still at least one round trip of potentially wasted capacity even then.

This specification defines a HTTP/2 frame type to allow clients to inform the server of their freshly cached contents using a Cuckoo-filter [Cuckoo] based digest. Servers can then use this to inform their choices of what to push to clients.

### 1.1. Notational Conventions

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].

## 2. The CACHE\_DIGEST Frame

The CACHE\_DIGEST frame type is 0xd (decimal 13).

```

+-----+-----+
|          Origin-Len (16)          | Origin? (\*)          ...
+-----+-----+
|                                Digest-Value? (\*)          ...
+-----+-----+
```

The CACHE\_DIGEST frame payload has the following fields:

**Origin-Len:** An unsigned, 16-bit integer indicating the length, in octets, of the Origin field.

**Origin:** A sequence of characters containing the ASCII serialization of an origin ([RFC6454], Section 6.2) that the Digest-Value applies to.

**Digest-Value:** A sequence of octets containing the digest as computed in Section 2.1.1 and Section 2.1.2.

The CACHE\_DIGEST frame defines the following flags:

- o **\*RESET\* (0x1):** When set, indicates that any and all cache digests for the applicable origin held by the recipient MUST be considered invalid.

- o \*COMPLETE\* (0x2): When set, indicates that the currently valid set of cache digests held by the server constitutes a complete representation of the cache's state regarding that origin.

## 2.1. Client Behavior

A CACHE\_DIGEST frame MUST be sent from a client to a server on stream 0, and conveys a digest of the contents of the client's cache for the indicated origin.

In typical use, a client will send one or more CACHE\_DIGESTs immediately after the first request on a connection for a given origin, on the same stream, because there is usually a short period of inactivity then, and servers can benefit most when they understand the state of the cache before they begin pushing associated assets (e.g., CSS, JavaScript and images). Clients MAY send CACHE\_DIGEST at other times.

If the cache's state is cleared, lost, or the client otherwise wishes the server to stop using previously sent CACHE\_DIGESTs, it can send a CACHE\_DIGEST with the RESET flag set.

When generating CACHE\_DIGEST, a client MUST NOT include stale-cached responses or responses whose URLs do not share origins [RFC6454] with the indicated origin. Clients MUST NOT send CACHE\_DIGEST frames on connections that are not authoritative (as defined in [RFC7540], 10.1) for the indicated origin.

When the CACHE\_DIGEST frames sent represent the complete set of stored responses, the last such frame SHOULD have a COMPLETE flag set, to indicate to the server that it has all relevant state. Note that for the purposes of COMPLETE, responses cached since the beginning of the connection or the last RESET flag on a CACHE\_DIGEST frame need not be included.

CACHE\_DIGEST has no defined meaning when sent from servers, and SHOULD be ignored by clients.

### 2.1.1. Creating a digest

Given the following inputs:

- o "P", an integer smaller than 256, that indicates the probability of a false positive that is acceptable, expressed as "1/2\\*\\*P".
- o "N", an integer that represents the number of entries - a prime number smaller than 2\*\*32

1. Let "f" be the number of bits per fingerprint, calculated as  $P + 3$
2. Let "b" be the bucket size, defined as 4.
3. Let "allocated" be the closest power of 2 that is larger than "N".
4. Let "bytes" be  $f * \text{"allocated"} * b / 8$  rounded up to the nearest integer
5. Add 5 to "bytes"
6. Allocate memory of "bytes" and set it to zero. Assign it to "digest-value".
7. Set the first byte to "P"
8. Set the second till fifth bytes to "N" in big endian form
9. Return the "digest-value".

Note: "allocated" is necessary due to the nature of the way Cuckoo filters are creating the secondary hash, by XORing the initial hash and the fingerprint's hash. The XOR operation means that secondary hash can pick an entry beyond the initial number of entries, up to the next power of 2. In order to avoid issues there, we allocate the table appropriately. For increased space efficiency, it is recommended that implementations pick a number of entries that's close to the next power of 2.

#### 2.1.2. Adding a URL to the Digest-Value

Given the following inputs:

- o "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
  - o "maxcount" - max number of cuckoo hops
  - o "digest-value"
1. Let "f" be the value of the first byte of "digest-value".
  2. Let "b" be the bucket size, defined as 4.
  3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.

4. Let "key" be the return value of Section 2.1.5 with "URL" as input.
5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.
6. Let "dest\_fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.
7. Let "h2" be the return value of Section 2.1.7 with "h1", "dest\_fingerprint" and "N" as inputs.
8. Let "h" be either "h1" or "h2", picked in random.
9. While "maxcount" is larger than zero:
  1. Let "position\_start" be  $40 + "h" * "f" * "b"$ .
  2. Let "position\_end" be  $"position\_start" + "f" * "b"$ .
  3. While "position\_start" < "position\_end":
    1. Let "bits" be "f" bits from "digest\_value" starting at "position\_start".
    2. If "bits" is all zeros, set "bits" to "dest\_fingerprint" and terminate these steps.
    3. Add "f" to "position\_start".
  4. Let "e" be a random number from 0 to "b".
  5. Subtract  $"f" * ("b" - "e")$  from "position\_start".
  6. Let "bits" be "f" bits from "digest\_value" starting at "position\_start".
  7. Let "fingerprint" be the value of bits, read as big endian.
  8. Set "bits" to "dest\_fingerprint".
  9. Set "dest\_fingerprint" to "fingerprint".
  10. Let "h" be Section 2.1.7 with "h", "dest\_fingerprint" and "N" as inputs.
  11. Subtract 1 from "maxcount".

10. Subtract "f" from "position\_start".
11. Let "fingerprint" be the "f" bits starting at "position\_start".
12. Let "h1" be "h"
13. Subtract 1 from "maxcount".
14. If "maxcount" is zero, return an error.
15. Go to step 7.

#### 2.1.3. Removing a URL to the Digest-Value

Given the following inputs:

- o "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
  - o "digest-value"
1. Let "f" be the value of the first byte of "digest-value".
  2. Let "b" be the bucket size, defined as 4.
  3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.
  4. Let "key" be the return value of Section 2.1.5 with "URL" as input.
  5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.
  6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.
  7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.
  8. Let "hashes" be an array containing "h1" and "h2".
  9. For each "h" in "hashes":
    1. Let "position\_start" be  $40 + "h" * "f" * "b"$ .
    2. Let "position\_end" be  $"position\_start" + "f" * "b"$ .

3. While "position\_start" < "position\_end":
  1. Let "bits" be "f" bits from "digest\_value" starting at "position\_start".
  2. If "bits" is "fingerprint", set "bits" to all zeros and terminate these steps.
  3. Add "f" to "position\_start".

#### 2.1.4. Computing a fingerprint value

Given the following inputs:

- o "key", an array of characters
  - o "f", an integer indicating the number of output bits
1. Let "hash-value" be the SHA-256 message digest [RFC6234] of "key", expressed as an integer.
  2. Let "h" be the number of bits in "hash-value"
  3. Let "fingerprint-value" be 0
  4. While "fingerprint-value" is 0 and "h" > "f":
    1. Let "fingerprint-value" be the "f" least significant bits of "hash-value".
    2. Let "hash-value" be the "h"- "f" most significant bits of "hash-value".
    3. Subtract "f" from "h".
  5. If "fingerprint-value" is 0, let "fingerprint-value" be 1.
  6. Return "fingerprint-value".

Note: Step 5 is to handle the extremely unlikely case where a SHA-256 digest of "key" is all zeros. The implications of it means that there's an infinitesimally larger probability of getting a "fingerprint-value" of 1 compared to all other values. This is not a problem for any practical purpose.



#### 2.1.5. Computing the key

Given the following inputs:

- o "URL", an array of characters
- 1. Let "key" be "URL" converted to an ASCII string by percent-encoding as appropriate [RFC3986].
- 2. Return "key"

#### 2.1.6. Computing a Hash Value

Given the following inputs:

- o "key", an array of characters.
- o "N", an integer

"hash-value" can be computed using the following algorithm:

1. Let "hash-value" be the SHA-256 message digest [RFC6234] of "key", truncated to 32 bits, expressed as an integer.
2. Return "hash-value" modulo N.

#### 2.1.7. Computing an Alternative Hash Value

Given the following inputs:

- o "hash1", an integer indicating the previous hash.
- o "fingerprint", an integer indicating the fingerprint value.
- o "N", an integer indicating the number of entries in the digest.
- 1. Let "fingerprint-string" be the value of "fingerprint" in base 10, expressed as a string.
- 2. Let "hash2" be the return value of Section 2.1.6 with "fingerprint-string" and "N" as inputs, XORed with "hash1".
- 3. Return "hash2".

## 2.2. Server Behavior

In typical use, a server will query (as per Section 2.2.1) the `CACHE_DIGESTS` received on a given connection to inform what it pushes to that client;

- o If a given URL has a match in a current `CACHE_DIGEST`, a complete response need not be pushed; The server MAY push a 304 response for that resource, indicating the client that it hasn't changed.
- o If a given URL has no match in any current `CACHE_DIGEST`, the client does not have a cached copy, and a complete response can be pushed.

Servers MAY use all `CACHE_DIGESTS` received for a given origin as current, as long as they do not have the `RESET` flag set; a `CACHE_DIGEST` frame with the `RESET` flag set MUST clear any previously stored `CACHE_DIGESTS` for its origin. Servers MUST treat an empty Digest-Value with a `RESET` flag set as effectively clearing all stored digests for that origin.

Clients are not likely to send updates to `CACHE_DIGEST` over the lifetime of a connection; it is expected that servers will separately track what cacheable responses have been sent previously on the same connection, using that knowledge in conjunction with that provided by `CACHE_DIGEST`.

Servers MUST ignore `CACHE_DIGEST` frames sent on a stream other than 0.

### 2.2.1. Querying the Digest for a Value

Given the following inputs:

- o "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234].
  - o "digest-value", an array of bits.
1. Let "f" be the value of the first byte of "digest-value".
  2. Let "b" be the bucket size, defined as 4.
  3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.
  4. Let "key" be the return value of Section 2.1.5 with "URL" as input.

5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.
  6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.
  7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.
  8. Let "hashes" be an array containing "h1" and "h2".
  9. For each "h" in "hashes":
    1. Let "position\_start" be  $40 + \text{"h"} * \text{"f"} * \text{"b"}$ .
    2. Let "position\_end" be  $\text{"position\_start"} + \text{"f"} * \text{"b"}$ .
    3. While  $\text{"position\_start"} < \text{"position\_end"}$ :
      1. Let "bits" be "f" bits from "digest\_value" starting at "position\_start".
      2. If "bits" is "fingerprint", return true
      3. Add "f" to "position\_start".
  10. Return false.
3. The SETTINGS\_SENDING\_CACHE\_DIGEST SETTINGS Parameter

A Client SHOULD notify its support for CACHE\_DIGEST frames by sending the SETTINGS\_SENDING\_CACHE\_DIGEST (0xXXX) SETTINGS parameter.

The value of the parameter is a bit-field of which the following bits are defined:

DIGEST\_PENDING (0x1): When set it indicates that the client has a digest to send, and the server may choose to wait for a digest in order to make server push decisions.

Rest of the bits MUST be ignored and MUST be left unset when sending.

The initial value of the parameter is zero (0x0) meaning that the client has no digest to send the server.

#### 4. The SETTINGS\_ACCEPT\_CACHE\_DIGEST SETTINGS Parameter

A server can notify its support for CACHE\_DIGEST frame by sending the SETTINGS\_ACCEPT\_CACHE\_DIGEST (0x7) SETTINGS parameter. If the server is tempted to making optimizations based on CACHE\_DIGEST frames, it SHOULD send the SETTINGS parameter immediately after the connection is established.

The value of the parameter is a bit-field of which the following bits are defined:

ACCEPT (0x1): When set, it indicates that the server is willing to make use of a digest of cached responses.

Rest of the bits MUST be ignored and MUST be left unset when sending.

The initial value of the parameter is zero (0x0) meaning that the server is not interested in seeing a CACHE\_DIGEST frame.

Some underlying transports allow the server's first flight of application data to reach the client at around the same time when the client sends its first flight data. When such transport (e.g., TLS 1.3 [I-D.ietf-tls-tls13] in full-handshake mode) is used, a client can postpone sending the CACHE\_DIGEST frame until it receives a SETTINGS\_ACCEPT\_CACHE\_DIGEST settings value.

When the underlying transport does not have such property (e.g., TLS 1.3 in 0-RTT mode), a client can reuse the settings value found in previous connections to that origin [RFC6454] to make assumptions.

#### 5. IANA Considerations

This document registers the following entry in the Permanent Message Headers Registry, as per [RFC3864]:

- o Header field name: Cache-Digest
- o Applicable protocol: http
- o Status: experimental
- o Author/Change controller: IESG
- o Specification document(s): [this document]

This document registers the following entry in the HTTP/2 Frame Type Registry, as per [RFC7540]:

- o Frame Type: CACHE\_DIGEST
- o Code: 0xd
- o Specification: [this document]

This document registers the following entry in the HTTP/2 Settings Registry, as per [RFC7540]:

- o Code: 0x7
- o Name: SETTINGS\_ACCEPT\_CACHE\_DIGEST
- o Initial Value: 0x0
- o Reference: [this document]

## 6. Security Considerations

The contents of a User Agent's cache can be used to re-identify or "fingerprint" the user over time, even when other identifiers (e.g., Cookies [RFC6265]) are cleared.

CACHE\_DIGEST allows such cache-based fingerprinting to become passive, since it allows the server to discover the state of the client's cache without any visible change in server behaviour.

As a result, clients MUST mitigate for this threat when the user attempts to remove identifiers (e.g., "clearing cookies"). This could be achieved in a number of ways; for example: by clearing the cache, by changing one or both of N and P, or by adding new, synthetic entries to the digest to change its contents.

TODO: discuss how effective the suggested mitigations actually would be.

Additionally, User Agents SHOULD NOT send CACHE\_DIGEST when in "privacy mode."

## 7. References

### 7.1. Normative References

- [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/rfc2119>>.

- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/info/rfc6454>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <<https://www.rfc-editor.org/info/rfc7232>>.
- [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/rfc7234>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

## 7.2. Informative References

- [Cuckoo] "Cuckoo Filter: Practically Better Than Bloom", n.d., <<https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf>>.
- [Fetch] "Fetch Standard", n.d., <<https://fetch.spec.whatwg.org/>>.
- [I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <<https://www.rfc-editor.org/info/rfc3864>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [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/rfc5234>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [Service-Workers]  
Russell, A., Song, J., Archibald, J., and M. Kruisselbrink, "Service Workers 1", W3C Working Draft WD-service-workers-1-20161011, October 2016, <<https://www.w3.org/TR/2016/WD-service-workers-1-20161011/>>.

#### Appendix A. Encoding the CACHE\_DIGEST frame as an HTTP Header

On some web browsers that support Service Workers [Service-Workers] but not Cache Digests (yet), it is possible to achieve the benefit of using Cache Digests by emulating the frame using HTTP Headers.

For the sake of interoperability with such clients, this appendix defines how a CACHE\_DIGEST frame can be encoded as an HTTP header named "Cache-Digest".

The definition uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with the list rule extension defined in [RFC7230], Section 7.

```
Cache-Digest = 1#digest-entity
digest-entity = digest-value *(OWS ";" OWS digest-flag)
digest-value = <Digest-Value encoded using base64url>
digest-flag = token
```

A Cache-Digest request header is defined as a list construct of cache-digest-entities. Each cache-digest-entity corresponds to a CACHE\_DIGEST frame.

Digest-Value is encoded using base64url [RFC4648], Section 5. Flags that are set are encoded as digest-flags by their names that are compared case-insensitively.

Origin is omitted in the header form. The value is implied from the value of the ":authority" pseudo header. Client MUST only send Cache-Digest headers containing digests that belong to the origin specified by the HTTP request.

The example below contains a digest of one resource and has only the "COMPLETE" flag set.

```
Cache-Digest: AfdA; complete
```

Clients MUST associate Cache-Digest headers to every HTTP request, since Fetch [Fetch] – the HTTP API supported by Service Workers – does not define the order in which the issued requests will be sent to the server nor guarantees that all the requests will be transmitted using a single HTTP/2 connection.

Also, due to the fact that any header that is supplied to Fetch is required to be end-to-end, there is an ambiguity in what a Cache-Digest header represents when a request is transmitted through a proxy. The header may represent the cache state of a client or that of a proxy, depending on how the proxy handles the header.

## Appendix B. Changes

### B.1. Since draft-ietf-httpbis-cache-digest-04

- o Remove ETag from the digest key calculations.
- o Add SETTINGS\_ prefix to parameter names.

### B.2. Since draft-ietf-httpbis-cache-digest-03

- o Yoav becomes an author; Mark steps down.

### B.3. Since draft-ietf-httpbis-cache-digest-02

- o Switch to Cuckoo Filter.

### B.4. Since draft-ietf-httpbis-cache-digest-01

- o Added definition of the Cache-Digest header.
- o Introduce ACCEPT\_CACHE\_DIGEST SETTINGS parameter.



- o Change intended status from Standard to Experimental.

B.5. Since draft-ietf-httpbis-cache-digest-00

- o Make the scope of a digest frame explicit and shift to stream 0.

Appendix C. Acknowledgements

+{:numbered="false"}

Thanks to Stefan Eissing for his suggestions.

Authors' Addresses

Kazuho Oku  
Fastly

Email: [kazuhooku@gmail.com](mailto:kazuhooku@gmail.com)

Yoav Weiss  
Akamai

Email: [yoav@yoav.ws](mailto:yoav@yoav.ws)  
URI: <https://blog.yoav.ws/>

HTTP Working Group  
Internet-Draft  
Intended status: Experimental  
Expires: May 1, 2018

K. Oku  
Fastly  
October 28, 2017

An HTTP Status Code for Indicating Hints  
draft-ietf-httpbis-early-hints-05

Abstract

This memo introduces an informational HTTP status code that can be used to convey hints that help a client make preparations for processing the final response.

Note to Readers

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

Working Group information can be found at <https://httpwg.github.io/> ; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/early-hints> .

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 May 1, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
1.1. Notational Conventions . . . . .	3
2. 103 Early Hints . . . . .	3
3. Security Considerations . . . . .	5
4. IANA Considerations . . . . .	6
5. References . . . . .	6
5.1. Normative References . . . . .	6
5.2. Informative References . . . . .	6
Appendix A. Changes . . . . .	6
A.1. Since draft-ietf-httpbis-early-hints-04 . . . . .	6
A.2. Since draft-ietf-httpbis-early-hints-03 . . . . .	7
A.3. Since draft-ietf-httpbis-early-hints-02 . . . . .	7
A.4. Since draft-ietf-httpbis-early-hints-01 . . . . .	7
A.5. Since draft-ietf-httpbis-early-hints-00 . . . . .	7
Appendix B. Acknowledgements . . . . .	7
Author's Address . . . . .	7

## 1. Introduction

It is common for HTTP responses to contain links to external resources that need to be fetched prior to their use; for example, rendering HTML by a Web browser. Having such links available to the client as early as possible helps to minimize perceived latency.

The "preload" ([Preload]) link relation can be used to convey such links in the Link header field of an HTTP response. However, it is not always possible for an origin server to generate the header block of a final response immediately after receiving a request. For example, the origin server might delegate a request to an upstream HTTP server running at a distant location, or the status code might depend on the result of a database query.

The dilemma here is that even though it is preferable for an origin server to send some header fields as soon as it receives a request, it cannot do so until the status code and the full header fields of the final HTTP response are determined.

HTTP/2 ([RFC7540]) server push can accelerate the delivery of resources, but only resources for which the server is authoritative. The other limitation of server push is that the response will be transmitted regardless of whether the client has the response cached. At the cost of spending one extra round-trip compared to server push in the worst case, delivering Link header fields in a timely fashion is more flexible and might consume less bandwidth.

This memo defines a status code for sending an informational response ([RFC7231], Section 6.2) that contains header fields that are likely to be included in the final response. A server can send the informational response containing some of the header fields to help the client start making preparations for processing the final response, and then run time-consuming operations to generate the final response. The informational response can also be used by an origin server to trigger HTTP/2 server push at a caching intermediary.

### 1.1. Notational Conventions

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].

## 2. 103 Early Hints

The 103 (Early Hints) informational status code indicates to the client that the server is likely to send a final response with the header fields included in the informational response.

Typically, a server will include the header fields sent in a 103 (Early Hints) response in the final response as well. However, there might be cases when this is not desirable, such as when the server learns that they are not correct before the final response is sent.

A client can speculatively evaluate the header fields included in a 103 (Early Hints) response while waiting for the final response. For example, a client might recognize a Link header field value containing the relation type "preload" and start fetching the target resource. However, these header fields only provide hints to the client; they do not replace the header fields on the final response.

Aside from performance optimizations, such evaluation of the 103 (Early Hints) response's header fields MUST NOT affect how the final response is processed. A client MUST NOT interpret the 103 (Early Hints) response header fields as if they applied to the informational response itself (e.g., as metadata about the 103 (Early Hints) response).

A server MAY use a 103 (Early Hints) response to indicate only some of the header fields that are expected to be found in the final response. A client SHOULD NOT interpret the nonexistence of a header field in a 103 (Early Hints) response as a speculation that the header field is unlikely to be part of the final response.

The following example illustrates a typical message exchange that involves a 103 (Early Hints) response.

Client request:

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

Server response:

```
HTTP/1.1 103 Early Hints
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

HTTP/1.1 200 OK
Date: Fri, 26 May 2017 10:02:11 GMT
Content-Length: 1234
Content-Type: text/html; charset=utf-8
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

<!doctype html>
[... rest of the response body is omitted from the example ...]
```

As is the case with any informational response, a server might emit more than one 103 (Early Hints) response prior to sending a final response. This can happen for example when a caching intermediary generates a 103 (Early Hints) response based on the header fields of a stale-cached response, then forwards a 103 (Early Hints) response and a final response that were sent from the origin server in response to a revalidation request.

A server MAY emit multiple 103 (Early Hints) responses with additional header fields as new information becomes available while the request is being processed. It does not need to repeat the fields that were already emitted, though it doesn't have to exclude them either. The client can consider any combination of header fields received in multiple 103 (Early Hints) responses when anticipating the list of header fields expected in the final response.

The following example illustrates a series of responses that a server might emit. In the example, the server uses two 103 (Early Hints) responses to notify the client that it is likely to send three Link header fields in the final response. Two of the three expected header fields are found in the final response. The other header field is replaced by another Link header field that contains a different value.

```
HTTP/1.1 103 Early Hints
Link: </main.css>; rel=preload; as=style

HTTP/1.1 103 Early Hints
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

HTTP/1.1 200 OK
Date: Fri, 26 May 2017 10:02:11 GMT
Content-Length: 1234
Content-Type: text/html; charset=utf-8
Link: </main.css>; rel=preload; as=style
Link: </newstyle.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

<!doctype html>
[... rest of the response body is omitted from the example ...]
```

### 3. Security Considerations

Some clients might have issues handling 103 (Early Hints), since informational responses are rarely used in reply to requests not including an Expect header field ([RFC7231], Section 5.1.1).

In particular, an HTTP/1.1 client that mishandles an informational response as a final response is likely to consider all responses to the succeeding requests sent over the same connection to be part of the final response. Such behavior might constitute a cross-origin information disclosure vulnerability in case the client multiplexes requests to different origins onto a single persistent connection.

Therefore, a server might refrain from sending Early Hints over HTTP/1.1 unless the client is known to handle informational responses correctly.

HTTP/2 clients are less likely to suffer from incorrect framing since handling of the response header fields does not affect how the end of the response body is determined.

#### 4. IANA Considerations

The HTTP Status Codes Registry will be updated with the following entry:

- o Code: 103
- o Description: Early Hints
- o Specification: [this document]

#### 5. References

##### 5.1. Normative References

- [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/rfc2119>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

##### 5.2. Informative References

- [Preload] Grigorik, I., "Preload", n.d., <<https://w3c.github.io/preload/>>.

#### Appendix A. Changes

##### A.1. Since draft-ietf-httpbis-early-hints-04

- o Clarified that the server is allowed to add headers not found in a 103 response to the final response.

- o Clarify client's behavior when it receives more than one 103 response.

A.2. Since draft-ietf-httpbis-early-hints-03

- o Removed statements that were either redundant or contradictory to RFC7230-7234.
- o Clarified what the server's expected behavior is.
- o Explain that a server might want to send more than one 103 response.
- o Editorial Changes.

A.3. Since draft-ietf-httpbis-early-hints-02

- o Editorial changes.
- o Added an example.

A.4. Since draft-ietf-httpbis-early-hints-01

- o Editorial changes.

A.5. Since draft-ietf-httpbis-early-hints-00

- o Forbid processing the headers of a 103 response as part of the informational response.

Appendix B. Acknowledgements

Thanks to Tatsuhiro Tsujikawa for coming up with the idea of sending the Link header fields using an informational response.

Author's Address

Kazuho Oku  
Fastly

Email: kazuhooku@gmail.com



HTTP  
Internet-Draft  
Intended status: Experimental  
Expires: June 12, 2019

E. Stark  
Google  
December 9, 2018

Expect-CT Extension for HTTP  
draft-ietf-httpbis-expect-ct-08

Abstract

This document defines a new HTTP header field named Expect-CT, which allows web host operators to instruct user agents to expect valid Signed Certificate Timestamps (SCTs) to be served on connections to these hosts. Expect-CT allows web host operators to discover misconfigurations in their Certificate Transparency deployments. Further, web host operators can use Expect-CT to ensure that, if a UA which supports Expect-CT accepts a misissued certificate, that certificate will be discoverable in Certificate Transparency logs.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list ([ietf-http-wg@w3.org](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> [1].

Working Group information can be found at <http://httpwg.github.io/> [2]; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/expect-ct> [3].

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 <https://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 June 12, 2019.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. Requirements Language . . . . .	4
1.2. Terminology . . . . .	4
2. Server and Client Behavior . . . . .	5
2.1. Response Header Field Syntax . . . . .	5
2.1.1. The report-uri Directive . . . . .	6
2.1.2. The enforce Directive . . . . .	7
2.1.3. The max-age Directive . . . . .	7
2.1.4. Examples . . . . .	7
2.2. Host Processing Model . . . . .	8
2.2.1. HTTP-over-Secure-Transport Request Type . . . . .	8
2.2.2. HTTP Request Type . . . . .	8
2.3. User Agent Processing Model . . . . .	8
2.3.1. Missing or Malformed Expect-CT Header Fields . . . . .	9
2.3.2. Expect-CT Header Field Processing . . . . .	9
2.3.3. Reporting . . . . .	11
2.4. Evaluating Expect-CT Connections for CT Compliance . . . . .	11
2.4.1. Skipping CT compliance checks . . . . .	12
3. Reporting Expect-CT Failure . . . . .	12
3.1. Generating a violation report . . . . .	12
3.2. Sending a violation report . . . . .	14
3.3. Receiving a violation report . . . . .	15
4. Usability Considerations . . . . .	16
5. Authoring Considerations . . . . .	16
6. Privacy Considerations . . . . .	16
7. Security Considerations . . . . .	17
7.1. Hostile header attacks . . . . .	17
7.2. Maximum max-age . . . . .	17
7.3. Amplification attacks . . . . .	18
8. IANA Considerations . . . . .	18
8.1. Header Field Registry . . . . .	18

8.2. Media Types Registry . . . . .	18
9. References . . . . .	19
9.1. Normative References . . . . .	19
9.2. Informative References . . . . .	21
9.3. URIs . . . . .	21
Appendix A. Changes . . . . .	21
A.1. Since -07 . . . . .	21
A.2. Since -06 . . . . .	22
A.3. Since -05 . . . . .	22
A.4. Since -04 . . . . .	22
A.5. Since -03 . . . . .	22
A.6. Since -02 . . . . .	22
A.7. Since -01 . . . . .	22
A.8. Since -00 . . . . .	22
Author's Address . . . . .	23

## 1. Introduction

This document defines a new HTTP header field that enables UAs to identify web hosts that expect the presence of Signed Certificate Timestamps (SCTs) [I-D.ietf-trans-rfc6962-bis] in subsequent Transport Layer Security (TLS) [RFC8446] connections.

Web hosts that serve the Expect-CT HTTP header field are noted by the UA as Known Expect-CT Hosts. The UA evaluates each connection to a Known Expect-CT Host for compliance with the UA's Certificate Transparency (CT) Policy. If the connection violates the CT Policy, the UA sends a report to a URI configured by the Expect-CT Host and/or fails the connection, depending on the configuration that the Expect-CT Host has chosen.

If misconfigured, Expect-CT can cause unwanted connection failures (for example, if a host deploys Expect-CT but then switches to a legitimate certificate that is not logged in Certificate Transparency logs, or if a web host operator believes their certificate to conform to all UAs' CT policies but is mistaken). Web host operators are advised to deploy Expect-CT with precautions, by using the reporting feature and gradually increasing the time interval during which the UA regards the host as a Known Expect-CT Host. These precautions can help web host operators gain confidence that their Expect-CT deployment is not causing unwanted connection failures.

Expect-CT is a trust-on-first-use (TOFU) mechanism. The first time a UA connects to a host, it lacks the information necessary to require SCTs for the connection. Thus, the UA will not be able to detect and thwart an attack on the UA's first connection to the host. Still, Expect-CT provides value by 1) allowing UAs to detect the use of unlogged certificates after the initial communication, and 2)

allowing web hosts to be confident that UAs are only trusting publicly-auditable certificates.

Expect-CT is similar to HSTS [RFC6797] and HPKP [RFC7469]. HSTS allows web sites to declare themselves accessible only via secure connections, and HPKP allows web sites to declare their cryptographic identifies. Similarly, Expect-CT allows web sites to declare themselves accessible only via connections that are compliant with CT policy.

This Expect-CT specification is compatible with [RFC6962] and [I-D.ietf-trans-rfc6962-bis], but not with future versions of Certificate Transparency. Expect-CT header fields will be ignore from web hosts which use future versions of Certificate Transparency, unless a future version of this document specifies how they should be processed.

### 1.1. Requirements Language

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.

### 1.2. Terminology

Terminology is defined in this section.

- o "Certificate Transparency Policy" is a policy defined by the UA concerning the number, sources, and delivery mechanisms of Signed Certificate Timestamps that are associated with TLS connections. The policy defines the properties of a connection that must be met in order for the UA to consider it CT-qualified.
- o "Certificate Transparency Qualified" describes a TLS connection for which the UA has determined that a sufficient quantity and quality of Signed Certificate Timestamps have been provided.
- o "CT-qualified" is an abbreviation for "Certificate Transparency Qualified".
- o "CT Policy" is an abbreviation for "Certificate Transparency Policy".
- o "Effective Expect-CT Date" is the time at which a UA observed a valid Expect-CT header field for a given host.

- o "Expect-CT Host" is a conformant host implementing the HTTP server aspects of Expect-CT. This means that an Expect-CT Host returns the "Expect-CT" HTTP response header field in its HTTP response messages sent over secure transport. The term "host" is equivalent to "server" in this specification.
- o "Known Expect-CT Host" is an Expect-CT Host that the UA has noted as such. See Section 2.3.2.1 for particulars.
- o UA is an acronym for "user agent". For the purposes of this specification, a UA is an HTTP client application typically actively manipulated by a user [RFC7230].
- o "Unknown Expect-CT Host" is an Expect-CT Host that the UA has not noted.

## 2. Server and Client Behavior

### 2.1. Response Header Field Syntax

The "Expect-CT" response header field is a new field defined in this specification. It is used by a server to indicate that UAs should evaluate connections to the host emitting the header field for CT compliance (Section 2.4).

Figure 1 describes the syntax (Augmented Backus-Naur Form) of the header field, using the grammar defined in [RFC5234] and the rules defined in Section 3.2 of [RFC7230]. The "#" ABNF extension is specified in Section 7 of [RFC7230].

```
Expect-CT           = 1#expect-ct-directive
expect-ct-directive = directive-name [ "=" directive-value ]
directive-name      = token
directive-value     = token / quoted-string
```

Figure 1: Syntax of the Expect-CT header field

The directives defined in this specification are described below. The overall requirements for directives are:

1. The order of appearance of directives is not significant.
2. A given directive MUST NOT appear more than once in a given header field. Directives are either optional or required, as stipulated in their definitions.
3. Directive names are case insensitive.

4. UAs MUST ignore any header fields containing directives, or other header field value data that do not conform to the syntax defined in this specification. In particular, UAs MUST NOT attempt to fix malformed header fields.
5. If a header field contains any directive(s) the UA does not recognize, the UA MUST ignore those directives.
6. If the Expect-CT header field otherwise satisfies the above requirements (1 through 5), and Expect-CT is not disabled for local policy reasons (as discussed in Section 2.4.1), the UA MUST process the directives it recognizes.

#### 2.1.1. The report-uri Directive

The OPTIONAL "report-uri" directive indicates the URI to which the UA SHOULD report Expect-CT failures (Section 2.4). The UA POSTs the reports to the given URI as described in Section 3.

The "report-uri" directive is REQUIRED to have a directive value, for which the syntax is defined in Figure 2.

report-uri-value = absolute-URI

Figure 2: Syntax of the report-uri directive value

"absolute-URI" is defined in Section 4.3 of [RFC3986].

UAs MUST ignore "report-uri"s that do not use the HTTPS scheme. UAs MUST check Expect-CT compliance when the host in the "report-uri" is a Known Expect-CT Host; similarly, UAs MUST apply HSTS [RFC6797] if the host in the "report-uri" is a Known HSTS Host.

UAs SHOULD make their best effort to report Expect-CT failures to the "report-uri", but they may fail to report in exceptional conditions. For example, if connecting to the "report-uri" itself incurs an Expect-CT failure or other certificate validation failure, the UA MUST cancel the connection. Similarly, if Expect-CT Host A sets a "report-uri" referring to Expect-CT Host B, and if B sets a "report-uri" referring to A, and if both hosts fail to comply to the UA's CT Policy, the UA SHOULD detect and break the loop by failing to send reports to and about those hosts.

Note that the report-uri need not necessarily be in the same Internet domain or web origin as the host being reported about. Hosts are in fact encouraged to use a separate host as the report-uri, so that CT failures on the Expect-CT host do not prevent reports from being sent.

UAs SHOULD limit the rate at which they send reports. For example, it is unnecessary to send the same report to the same "report-uri" more than once in the same web browsing session.

#### 2.1.2. The enforce Directive

The OPTIONAL "enforce" directive is a valueless directive that, if present (i.e., it is "asserted"), signals to the UA that compliance to the CT Policy should be enforced (rather than report-only) and that the UA should refuse future connections that violate its CT Policy. When both the "enforce" directive and "report-uri" directive (as defined in Figure 2) are present, the configuration is referred to as an "enforce-and-report" configuration, signalling to the UA both that compliance to the CT Policy should be enforced and that violations should be reported.

#### 2.1.3. The max-age Directive

The "max-age" directive specifies the number of seconds after the reception of the Expect-CT header field during which the UA SHOULD regard the host from whom the message was received as a Known Expect-CT Host.

If a response contains an "Expect-CT" header field, then the response MUST contain an "Expect-CT" header field with a "max-age" directive. (A "max-age" directive need not appear in every "Expect-CT" header field in the response.) The "max-age" directive is REQUIRED to have a directive value, for which the syntax (after quoted-string unescaping, if necessary) is defined in Figure 3.

```
max-age-value = delta-seconds
delta-seconds = 1 *DIGIT
```

Figure 3: Syntax of the max-age directive value

"delta-seconds" is used as defined in Section 1.2.1 of [RFC7234].

#### 2.1.4. Examples

The following three examples demonstrate valid Expect-CT response header fields (where the second splits the directives into two field instances):

```
Expect-CT: max-age=86400, enforce  
  
Expect-CT: max-age=86400,enforce  
Expect-CT: report-uri="https://foo.example/report"  
  
Expect-CT: max-age=86400,report-uri="https://foo.example/report"
```

Figure 4: Examples of valid Expect-CT response header fields

## 2.2. Host Processing Model

This section describes the processing model that Expect-CT Hosts implement. The model has 2 parts: (1) the processing rules for HTTP request messages received over a secure transport (e.g., authenticated, non-anonymous TLS); and (2) the processing rules for HTTP request messages received over non-secure transports, such as TCP.

### 2.2.1. HTTP-over-Secure-Transport Request Type

An Expect-CT Host includes an Expect-CT header field in its response. The header field **MUST** satisfy the grammar specified in Section 2.1.

Establishing a given host as an Expect-CT Host, in the context of a given UA, is accomplished as follows:

1. Over the HTTP protocol running over secure transport, by correctly returning (per this specification) a valid Expect-CT header field to the UA.
2. Through other mechanisms, such as a client-side preloaded Expect-CT Host list.

### 2.2.2. HTTP Request Type

Expect-CT Hosts **SHOULD NOT** include the Expect-CT header field in HTTP responses conveyed over non-secure transport.

## 2.3. User Agent Processing Model

The UA processing model relies on parsing domain names. Note that internationalized domain names **SHALL** be canonicalized by the UA according to the scheme in Section 10 of [RFC6797].

The UA stores Known Expect-CT Hosts and their associated Expect-CT directives. This data is collectively known as a host's "Expect-CT" metadata".



### 2.3.1. Missing or Malformed Expect-CT Header Fields

If an HTTP response does not include an Expect-CT header field that conforms to the grammar specified in Section 2.1, then the UA MUST NOT update any Expect-CT metadata.

### 2.3.2. Expect-CT Header Field Processing

If the UA receives an HTTP response over a secure transport that includes an Expect-CT header field conforming to the grammar specified in Section 2.1, the UA MUST evaluate the connection on which the header field was received for compliance with the UA's CT Policy, and then process the Expect-CT header field as follows. UAs MUST ignore any Expect-CT header field received in an HTTP response conveyed over non-secure transport.

If the connection does not comply with the UA's CT Policy (i.e., the connection is not CT-qualified), then the UA MUST NOT update any Expect-CT metadata. If the header field includes a "report-uri" directive, the UA SHOULD send a report to the specified "report-uri" (Section 2.3.3).

If the connection complies with the UA's CT Policy (i.e., the connection is CT-qualified), then the UA MUST either:

- o Note the host as a Known Expect-CT Host if it is not already so noted (see Section 2.3.2.1), or
- o Update the UA's cached information for the Known Expect-CT Host if the "enforce", "max-age", or "report-uri" header field value directives convey information different from that already maintained by the UA. If the "max-age" directive has a value of 0, the UA MUST remove its cached Expect-CT information if the host was previously noted as a Known Expect-CT Host, and MUST NOT note this host as a Known Expect-CT Host if it is not already noted.

If a UA receives an Expect-CT header field over a CT-compliant connection which uses a version of Certificate Transparency other than [RFC6962] or [I-D.ietf-trans-rfc6962-bis], the UA MUST ignore the Expect-CT header field and clear any Expect-CT metadata associated with the host.

#### 2.3.2.1. Noting Expect-CT

Upon receipt of the Expect-CT response header field over an error-free TLS connection (with X.509 certificate chain validation as described in [RFC5280], as well as the validation described in Section 2.4), the UA MUST note the host as a Known Expect-CT Host,

storing the host's domain name and its associated Expect-CT directives in non-volatile storage.

To note a host as a Known Expect-CT Host, the UA MUST set its Expect-CT metadata in its Known Expect-CT Host cache (as specified in Section 2.3.2.2, using the metadata given in the most recently received valid Expect-CT header field.

For forward compatibility, the UA MUST ignore any unrecognized Expect-CT header field directives, while still processing those directives it does recognize. Section 2.1 specifies the directives "enforce", "max-age", and "report-uri", but future specifications and implementations might use additional directives.

#### 2.3.2.2. Storage Model

If the substring matching the host production from the Request-URI (of the message to which the host responded) does not exactly match an existing Known Expect-CT Host's domain name, per the matching procedure for a Congruent Match specified in Section 8.2 of [RFC6797], then the UA MUST add this host to the Known Expect-CT Host cache. The UA caches:

- o the Expect-CT Host's domain name,
- o whether the "enforce" directive is present
- o the Effective Expiration Date, which is the Effective Expect-CT Date plus the value of the "max-age" directive. Alternatively, the UA MAY cache enough information to calculate the Effective Expiration Date. The Effective Expiration Date is calculated from when the UA observed the Expect-CT header field and is independent of when the response was generated.
- o the value of the "report-uri" directive, if present.

If any other metadata from optional or future Expect-CT header directives are present in the Expect-CT header field, and the UA understands them, the UA MAY note them as well.

UAs MAY set an upper limit on the value of max-age, so that UAs that have noted erroneous Expect-CT hosts (whether by accident or due to attack) have some chance of recovering over time. If the server sets a max-age greater than the UA's upper limit, the UA may behave as if the server set the max-age to the UA's upper limit. For example, if the UA caps max-age at 5,184,000 seconds (60 days), and an Expect-CT Host sets a max-age directive of 90 days in its Expect-CT header field, the UA may behave as if the max-age were effectively 60 days.

(One way to achieve this behavior is for the UA to simply store a value of 60 days instead of the 90-day value provided by the Expect-CT host.)

### 2.3.3. Reporting

If the UA receives, over a secure transport, an HTTP response that includes an Expect-CT header field with a "report-uri" directive, and the connection does not comply with the UA's CT Policy (i.e., the connection is not CT-qualified), and the UA has not already sent an Expect-CT report for this connection, then the UA SHOULD send a report to the specified "report-uri" as specified in Section 3.

### 2.4. Evaluating Expect-CT Connections for CT Compliance

When a UA sets up a TLS connection, the UA determines whether the host is a Known Expect-CT Host according to its Known Expect-CT Host cache. An Expect-CT Host is "expired" if the effective expiration date refers to a date in the past. The UA MUST ignore any expired Expect-CT Hosts in its cache and not treat such hosts as Known Expect-CT hosts.

When a UA connects to a Known Expect-CT Host using a TLS connection, if the TLS connection has no errors, then the UA will apply an additional correctness check: compliance with a CT Policy. A UA should evaluate compliance with its CT Policy whenever connecting to a Known Expect-CT Host. However, the check can be skipped for local policy reasons (as discussed in Section 2.4.1), or in the event that other checks cause the UA to terminate the connection before CT compliance is evaluated. For example, a Public Key Pinning failure [RFC7469] could cause the UA to terminate the connection before CT compliance is checked. Similarly, if the UA terminates the connection due to an Expect-CT failure, this could cause the UA to skip subsequent correctness checks. When the CT compliance check is skipped or bypassed, Expect-CT reports (Section 3) will not be sent.

When CT compliance is evaluated for a Known Expect-CT Host, the UA MUST evaluate compliance when setting up the TLS session, before beginning an HTTP conversation over the TLS channel.

If a connection to a Known Expect-CT Host violates the UA's CT policy (i.e., the connection is not CT-qualified), and if the Known Expect-CT Host's Expect-CT metadata indicates an "enforce" configuration, the UA MUST treat the CT compliance failure as an error. The UA MAY allow the user to bypass the error, unless connection errors should have no user recourse due to other policies in effect (such as HSTS, as described in Section 12.1 of [RFC6797]).

If a connection to a Known Expect-CT Host violates the UA's CT policy, and if the Known Expect-CT Host's Expect-CT metadata includes a "report-uri", the UA SHOULD send an Expect-CT report to that "report-uri" (Section 3).

#### 2.4.1. Skipping CT compliance checks

It is acceptable for a UA to skip CT compliance checks for some hosts according to local policy. For example, a UA MAY disable CT compliance checks for hosts whose validated certificate chain terminates at a user-defined trust anchor, rather than a trust anchor built-in to the UA (or underlying platform).

If the UA does not evaluate CT compliance, e.g., because the user has elected to disable it, or because a presented certificate chain chains up to a user-defined trust anchor, UAs SHOULD NOT send Expect-CT reports.

### 3. Reporting Expect-CT Failure

When the UA attempts to connect to a Known Expect-CT Host and the connection is not CT-qualified, the UA SHOULD report Expect-CT failures to the "report-uri", if any, in the Known Expect-CT Host's Expect-CT metadata.

When the UA receives an Expect-CT response header field over a connection that is not CT-qualified, if the UA has not already sent an Expect-CT report for this connection, then the UA SHOULD report Expect-CT failures to the configured "report-uri", if any.

#### 3.1. Generating a violation report

To generate a violation report object, the UA constructs a JSON [RFC8259] object with the following keys and values:

- o "date-time": the value for this key indicates the UTC time that the UA observed the CT compliance failure. The value is a string formatted according to Section 5.6, "Internet Date/Time Format", of [RFC3339].
- o "hostname": the value is the hostname to which the UA made the original request that failed the CT compliance check. The value is provided as a string.
- o "port": the value is the port to which the UA made the original request that failed the CT compliance check. The value is provided as an integer.

- o "scheme": the value is the scheme with which the UA made the original request that failed the CT compliance check. The value is provided as a string. This key is optional and is assumed to be "https" if not present.
- o "effective-expiration-date": the value indicates the Effective Expiration Date (see Section 2.3.2.2) for the Expect-CT Host that failed the CT compliance check, in UTC. The value is provided as a string formatted according to Section 5.6 of [RFC3339] ("Internet Date/Time Format").
- o "served-certificate-chain": the value is the certificate chain as served by the Expect-CT Host during TLS session setup. The value is provided as an array of strings, which MUST appear in the order that the certificates were served; each string in the array is the Privacy-Enhanced Mail (PEM) representation of each X.509 certificate as described in [RFC7468].
- o "validated-certificate-chain": the value is the certificate chain as constructed by the UA during certificate chain verification. (This may differ from the value of the "served-certificate-chain" key.) The value is provided as an array of strings, which MUST appear in the order matching the chain that the UA validated; each string in the array is the Privacy-Enhanced Mail (PEM) representation of each X.509 certificate as described in [RFC7468]. The first certificate in the chain represents the end-entity certificate being verified. UAs that build certificate chains in more than one way during the validation process SHOULD send the last chain built.
- o "scts": the value represents the SCTs (if any) that the UA received for the Expect-CT host and their validation statuses. The value is provided as an array of JSON objects. The SCTs may appear in any order. Each JSON object in the array has the following keys:
  - \* A "version" key, with an integer value. The UA MUST set this value to "1" if the SCT is in the format defined in Section 3.2 of [RFC6962] and "2" if it is in the format defined in Section 4.5 of [I-D.ietf-trans-rfc6962-bis].
  - \* The "status" key, with a string value that the UA MUST set to one of the following values: "unknown" (indicating that the UA does not have or does not trust the public key of the log from which the SCT was issued), "valid" (indicating that the UA successfully validated the SCT as described in Section 5.2 of [RFC6962] or Section 8.2.3 of [I-D.ietf-trans-rfc6962-bis]), or

"invalid" (indicating that the SCT validation failed because of a bad signature or an invalid timestamp).

- \* The "source" key, with a string value that indicates from where the UA obtained the SCT, as defined in Section 3 of [RFC6962] and Section 6 of [I-D.ietf-trans-rfc6962-bis]. The UA MUST set the value to one of "tls-extension", "ocsp", or "embedded". These correspond to the three methods of delivering SCTs in the TLS handshake that are described in Section 3.3 of [RFC6962].
- \* The "serialized\_sct" key, with a string value. If the value of the "version" key is "1", the UA MUST set this value to the base64 encoded [RFC4648] serialized "SignedCertificateTimestamp" structure from Section 3.2 of [RFC6962]. The base64 encoding is defined in Section 4 of [RFC4648]. If the value of the "version" key is "2", the UA MUST set this value to the base64 encoded [RFC4648] serialized "TransItem" structure representing the SCT, as defined in Section 4.5 of [I-D.ietf-trans-rfc6962-bis].
- o "failure-mode": the value indicates whether the Expect-CT report was triggered by an Expect-CT policy in enforce or report-only mode. The value is provided as a string. The UA MUST set this value to "enforce" if the Expect-CT metadata indicates an "enforce" configuration, and "report-only" otherwise.
- o "test-report": the value is set to true if the report is being sent by a testing client to verify that the report server behaves correctly. The value is provided as a boolean, and MUST be set to true if the report serves to test the server's behavior and can be discarded.

### 3.2. Sending a violation report

The UA SHOULD report Expect-CT failures for Known Expect-CT Hosts: that is, when a connection to a Known Expect-CT Host does not comply with the UA's CT Policy and the host's Expect-CT metadata contains a "report-uri".

Additionally, the UA SHOULD report Expect-CT failures for hosts for which it does not have any stored Expect-CT metadata. That is, when the UA connects to a host and receives an Expect-CT header field which contains the "report-uri" directive, the UA SHOULD report an Expect-CT failure if the the connection does not comply with the UA's CT Policy.

The steps to report an Expect-CT failure are as follows.

1. Prepare a JSON object "report object" with the single key "expect-ct-report", whose value is the result of generating a violation report object as described in Section 3.1.
2. Let "report body" be the JSON stringification of "report object".
3. Let "report-uri" be the value of the "report-uri" directive in the Expect-CT header field.
4. Send an HTTP POST request to "report-uri" with a "Content-Type" header field of "application/expect-ct-report+json", and an entity body consisting of "report body".

The UA MAY perform other operations as part of sending the HTTP POST request, for example sending a CORS preflight as part of [FETCH].

Future versions of this specification may need to modify or extend the Expect-CT report format. They may do so by defining a new top-level key to contain the report, replacing the "expect-ct-report" key. Section 3.3 defines how report servers should handle report formats that they do not support.

### 3.3. Receiving a violation report

Upon receiving an Expect-CT violation report, the report server MUST respond with a 2xx (Successful) status code if it can parse the request body as valid JSON, the report conforms to the format described in Section 3.1, and it recognizes the scheme, hostname, and port in the "scheme", "hostname", and "port" fields of the report. If the report body cannot be parsed, or the report does not conform to the format described in Section 3.1, or the report server does not expect to receive reports for the scheme, hostname, or port in the report, then the report server MUST respond with a 400 Bad Request status code.

As described in Section 3.2, future versions of this specification may define new report formats that are sent with a different top-level key. If the report server does not recognize the report format, the report server MUST respond with a 501 Not Implemented status code.

If the report's "test-report" key is set to true, the server MAY discard the report without further processing but MUST still return a 2xx (Successful) status code. If the "test-report" key is absent or set to false, the server SHOULD store the report for processing and analysis by the owner of the Expect-CT Host.

#### 4. Usability Considerations

When the UA detects a Known Expect-CT Host in violation of the UA's CT Policy, end users will experience denials of service. It is advisable for UAs to explain to users why they cannot access the Expect-CT Host, e.g., in a user interface that explains that the host's certificate cannot be validated.

#### 5. Authoring Considerations

Expect-CT could be specified as a TLS extension or X.509 certificate extension instead of an HTTP response header field. Using an HTTP header field as the mechanism for Expect-CT introduces a layering mismatch: for example, the software that terminates TLS and validates Certificate Transparency information might know nothing about HTTP. Nevertheless, an HTTP header field was chosen primarily for ease of deployment. In practice, deploying new certificate extensions requires certificate authorities to support them, and new TLS extensions require server software updates, including possibly to servers outside of the site owner's direct control (such as in the case of a third-party CDN). Ease of deployment is a high priority for Expect-CT because it is intended as a temporary transition mechanism for user agents that are transitioning to universal Certificate Transparency requirements.

#### 6. Privacy Considerations

Expect-CT can be used to infer what Certificate Transparency policy a UA is using, by attempting to retrieve specially-configured websites which pass one user agents' policies but not another's. Note that this consideration is true of UAs which enforce CT policies without Expect-CT as well.

Additionally, reports submitted to the "report-uri" could reveal information to a third party about which webpage is being accessed and by which IP address, by using individual "report-uri" values for individually-tracked pages. This information could be leaked even if client-side scripting were disabled.

Implementations store state about Known Expect-CT Hosts, and hence which domains the UA has contacted. Implementations may choose to not store this state subject to local policy (e.g., in the private browsing mode of a web browser).

Violation reports, as noted in Section 3, contain information about the certificate chain that has violated the CT policy. In some cases, such as organization-wide compromise of the end-to-end security of TLS, this may include information about the interception



tools and design used by the organization that the organization would otherwise prefer not be disclosed.

Because Expect-CT causes remotely-detectable behavior, it's advisable that UAs offer a way for privacy-sensitive end users to clear currently noted Expect-CT hosts, and allow users to query the current state of Known Expect-CT Hosts.

## 7. Security Considerations

### 7.1. Hostile header attacks

When UAs support the Expect-CT header field, it becomes a potential vector for hostile header attacks against site owners. If a site owner uses a certificate issued by a certificate authority which does not embed SCTs nor serve SCTs via OCSP or TLS extension, a malicious server operator or attacker could temporarily reconfigure the host to comply with the UA's CT policy, and add the Expect-CT header field in enforcing mode with a long "max-age". Implementing user agents would note this as an Expect-CT Host (see Section 2.3.2.1). After having done this, the configuration could then be reverted to not comply with the CT policy, prompting failures. Note this scenario would require the attacker to have substantial control over the infrastructure in question, being able to obtain different certificates, change server software, or act as a man-in-the-middle in connections.

Site operators can mitigate this situation by one of: reconfiguring their web server to transmit SCTs using the TLS extension defined in Section 6.5 of [I-D.ietf-trans-rfc6962-bis], obtaining a certificate from an alternative certificate authority which provides SCTs by one of the other methods, or by waiting for the user agents' persisted notation of this as an Expect-CT host to reach its "max-age". User agents may choose to implement mechanisms for users to cure this situation, as noted in Section 4.

### 7.2. Maximum max-age

There is a security trade-off in that low maximum values provide a narrow window of protection for users that visit the Known Expect-CT Host only infrequently, while high maximum values might result in a denial of service to a UA in the event of a hostile header attack, or simply an error on the part of the site-owner.

There is probably no ideal maximum for the "max-age" directive. Since Expect-CT is primarily a policy-expansion and investigation technology rather than an end-user protection, a value on the order

of 30 days (2,592,000 seconds) may be considered a balance between these competing security concerns.

### 7.3. Amplification attacks

Another kind of hostile header attack uses the "report-uri" mechanism on many hosts not currently exposing SCTs as a method to cause a denial-of-service to the host receiving the reports. If some highly-trafficked websites emitted a non-enforcing Expect-CT header field with a "report-uri", implementing UAs' reports could flood the reporting host. It is noted in Section 2.1.1 that UAs should limit the rate at which they emit reports, but an attacker may alter the Expect-CT header's fields to induce UAs to submit different reports to different URIs to still cause the same effect.

## 8. IANA Considerations

### 8.1. Header Field Registry

This document registers the "Expect-CT" header field in the "Permanent Message Header Field Names" registry located at <https://www.iana.org/assignments/message-headers> [4].

Header field name: Expect-CT

Applicable protocol: http

Status: experimental

Author/Change controller: IETF

Specification document(s): This document

Related information: (empty)

### 8.2. Media Types Registry

The MIME media type for Expect-CT violation reports is "application/expect-ct-report+json" (which uses the suffix established in [RFC6839]).

Type name: application

Subtype name: expect-ct-report+json

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See Section 7

Interoperability considerations: n/a

Published specification: This document

Applications that use this media type: UAs that implement  
Certificate Transparency compliance checks and reporting

Additional information:

Deprecated alias names for this type: n/a

Magic number(s): n/a

File extension(s): n/a

Macintosh file type code(s): n/a

Person & email address to contact for further information: Emily  
Stark (estark@google.com)

Intended usage: COMMON

Restrictions on usage: none

Author: Emily Stark (estark@google.com)

Change controller: IETF

## 9. References

### 9.1. Normative References

- [I-D.ietf-trans-rfc6962-bis]  
Laurie, B., Langley, A., Kasper, E., Messeri, E., and R.  
Stradling, "Certificate Transparency Version 2.0", draft-  
ietf-trans-rfc6962-bis-30 (work in progress), November  
2018.
- [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/rfc2119>>.

- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [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/rfc5234>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <<https://www.rfc-editor.org/info/rfc6797>>.
- [RFC6839] Hansen, T. and A. Melnikov, "Additional Media Type Structured Syntax Suffixes", RFC 6839, DOI 10.17487/RFC6839, January 2013, <<https://www.rfc-editor.org/info/rfc6839>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [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/rfc7234>>.

- [RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <<https://www.rfc-editor.org/info/rfc7468>>.
- [RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <<https://www.rfc-editor.org/info/rfc7469>>.
- [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/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

## 9.2. Informative References

- [FETCH] WHATWG, "Fetch - Living Standard", n.d., <<https://fetch.spec.whatwg.org>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

## 9.3. URIs

- [1] <https://lists.w3.org/Archives/Public/ietf-http-wg/>
- [2] <http://httpwg.github.io/>
- [3] <https://github.com/httpwg/http-extensions/labels/expect-ct>
- [4] <https://www.iana.org/assignments/message-headers>

## Appendix A. Changes

### A.1. Since -07

- o Editorial changes
- o Specify that the end-entity certificate appears first in the "validated-certificate-chain" field of an Expect-CT report.
- o Define how report format can be extended by future versions of this specification.

- o Add optional "scheme" key to report format.
- o Specify exact status codes for report server errors.
- o Limit report-uris to HTTPS only.
- o Note that this version of Expect-CT is only compatible with RFC 6962 and 6962-bis, not any future versions of CT.

A.2. Since -06

- o Editorial changes

A.3. Since -05

- o Remove SHOULD requirement that UAs disallow certificate error overrides for Known Expect-CT Hosts.
- o Remove restriction that Expect-CT Hosts cannot be IP addresses.
- o Editorial changes

A.4. Since -04

- o Editorial changes

A.5. Since -03

- o Editorial changes

A.6. Since -02

- o Add concept of test reports and specify that servers must respond with 2xx status codes to valid reports.
- o Add "failure-mode" key to reports to allow report servers to distinguish report-only from enforced failures.

A.7. Since -01

- o Change SCT reporting format to support both RFC 6962 and 6962-bis SCTs.

A.8. Since -00

- o Editorial changes

- o Change Content-Type header of reports to 'application/expect-ct-report+json'
- o Update header field syntax to match convention (issue #327)
- o Reference RFC 6962-bis instead of RFC 6962

Author's Address

Emily Stark  
Google

Email: [estark@google.com](mailto:estark@google.com)

HTTP  
Internet-Draft  
Intended status: Standards Track  
Expires: December 5, 2020

M. Nottingham  
Fastly  
P-H. Kamp  
The Varnish Cache Project  
June 3, 2020

Structured Field Values for HTTP  
draft-ietf-httpbis-header-structure-19

Abstract

This document describes a set of data types and associated algorithms that are intended to make it easier and safer to define and handle HTTP header and trailer fields, known as "Structured Fields", "Structured Headers", or "Structured Trailers". It is intended for use by specifications of new HTTP fields that wish to use a common syntax that is more restrictive than traditional HTTP field values.

Note to Readers

\_RFC EDITOR: please remove this section before publication\_

Discussion of this draft takes place on the HTTP working group mailing list ([ietf-http-wg@w3.org](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> [1].

Working Group information can be found at <https://httpwg.github.io/> [2]; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/header-structure> [3].

Tests for implementations are collected at <https://github.com/httpwg/structured-field-tests> [4].

Implementations are tracked at <https://github.com/httpwg/wiki/wiki/Structured-Headers> [5].

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 <https://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 December 5, 2020.

#### Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

#### Table of Contents

1. Introduction . . . . .	4
1.1. Intentionally Strict Processing . . . . .	4
1.2. Notational Conventions . . . . .	5
2. Defining New Structured Fields . . . . .	5
3. Structured Data Types . . . . .	8
3.1. Lists . . . . .	9
3.1.1. Inner Lists . . . . .	9
3.1.2. Parameters . . . . .	10
3.2. Dictionaries . . . . .	11
3.3. Items . . . . .	12
3.3.1. Integers . . . . .	13
3.3.2. Decimals . . . . .	13
3.3.3. Strings . . . . .	14
3.3.4. Tokens . . . . .	15
3.3.5. Byte Sequences . . . . .	15
3.3.6. Booleans . . . . .	15
4. Working With Structured Fields in HTTP . . . . .	16
4.1. Serializing Structured Fields . . . . .	16
4.1.1. Serializing a List . . . . .	16
4.1.2. Serializing a Dictionary . . . . .	18
4.1.3. Serializing an Item . . . . .	19
4.1.4. Serializing an Integer . . . . .	20
4.1.5. Serializing a Decimal . . . . .	20
4.1.6. Serializing a String . . . . .	21

4.1.7. Serializing a Token . . . . .	22
4.1.8. Serializing a Byte Sequence . . . . .	22
4.1.9. Serializing a Boolean . . . . .	22
4.2. Parsing Structured Fields . . . . .	23
4.2.1. Parsing a List . . . . .	24
4.2.2. Parsing a Dictionary . . . . .	26
4.2.3. Parsing an Item . . . . .	27
4.2.4. Parsing an Integer or Decimal . . . . .	29
4.2.5. Parsing a String . . . . .	30
4.2.6. Parsing a Token . . . . .	31
4.2.7. Parsing a Byte Sequence . . . . .	32
4.2.8. Parsing a Boolean . . . . .	33
5. IANA Considerations . . . . .	33
6. Security Considerations . . . . .	33
7. References . . . . .	33
7.1. Normative References . . . . .	33
7.2. Informative References . . . . .	34
7.3. URIs . . . . .	35
Appendix A. Frequently Asked Questions . . . . .	35
A.1. Why not JSON? . . . . .	35
Appendix B. Implementation Notes . . . . .	36
Appendix C. Changes . . . . .	36
C.1. Since draft-ietf-httpbis-header-structure-18 . . . . .	37
C.2. Since draft-ietf-httpbis-header-structure-17 . . . . .	37
C.3. Since draft-ietf-httpbis-header-structure-16 . . . . .	37
C.4. Since draft-ietf-httpbis-header-structure-15 . . . . .	37
C.5. Since draft-ietf-httpbis-header-structure-14 . . . . .	38
C.6. Since draft-ietf-httpbis-header-structure-13 . . . . .	38
C.7. Since draft-ietf-httpbis-header-structure-12 . . . . .	39
C.8. Since draft-ietf-httpbis-header-structure-11 . . . . .	39
C.9. Since draft-ietf-httpbis-header-structure-10 . . . . .	39
C.10. Since draft-ietf-httpbis-header-structure-09 . . . . .	39
C.11. Since draft-ietf-httpbis-header-structure-08 . . . . .	40
C.12. Since draft-ietf-httpbis-header-structure-07 . . . . .	40
C.13. Since draft-ietf-httpbis-header-structure-06 . . . . .	41
C.14. Since draft-ietf-httpbis-header-structure-05 . . . . .	41
C.15. Since draft-ietf-httpbis-header-structure-04 . . . . .	41
C.16. Since draft-ietf-httpbis-header-structure-03 . . . . .	41
C.17. Since draft-ietf-httpbis-header-structure-02 . . . . .	41
C.18. Since draft-ietf-httpbis-header-structure-01 . . . . .	42
C.19. Since draft-ietf-httpbis-header-structure-00 . . . . .	42
Acknowledgements . . . . .	42
Authors' Addresses . . . . .	42

## 1. Introduction

Specifying the syntax of new HTTP header (and trailer) fields is an onerous task; even with the guidance in Section 8.3.1 of [RFC7231], there are many decisions - and pitfalls - for a prospective HTTP field author.

Once a field is defined, bespoke parsers and serializers often need to be written, because each field value has slightly different handling of what looks like common syntax.

This document introduces a set of common data structures for use in definitions of new HTTP field values to address these problems. In particular, it defines a generic, abstract model for them, along with a concrete serialization for expressing that model in HTTP [RFC7230] header and trailer fields.

A HTTP field that is defined as a "Structured Header" or "Structured Trailer" (if the field can be either, it is a "Structured Field") uses the types defined in this specification to define its syntax and basic handling rules, thereby simplifying both its definition by specification writers and handling by implementations.

Additionally, future versions of HTTP can define alternative serializations of the abstract model of these structures, allowing fields that use that model to be transmitted more efficiently without being redefined.

Note that it is not a goal of this document to redefine the syntax of existing HTTP fields; the mechanisms described herein are only intended to be used with fields that explicitly opt into them.

Section 2 describes how to specify a Structured Field.

Section 3 defines a number of abstract data types that can be used in Structured Fields.

Those abstract types can be serialized into and parsed from HTTP field values using the algorithms described in Section 4.

### 1.1. Intentionally Strict Processing

This specification intentionally defines strict parsing and serialization behaviors using step-by-step algorithms; the only error handling defined is to fail the operation altogether.

It is designed to encourage faithful implementation and therefore good interoperability. Therefore, an implementation that tried to be

helpful by being more tolerant of input would make interoperability worse, since that would create pressure on other implementations to implement similar (but likely subtly different) workarounds.

In other words, strict processing is an intentional feature of this specification; it allows non-conformant input to be discovered and corrected by the producer early, and avoids both interoperability and security issues that might otherwise result.

Note that as a result of this strictness, if a field is appended to by multiple parties (e.g., intermediaries, or different components in the sender), an error in one party's value is likely to cause the entire field value to fail parsing.

## 1.2. Notational Conventions

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.

This document uses algorithms to specify parsing and serialization behaviors, and the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] to illustrate expected syntax in HTTP header fields. In doing so, it uses the VCHAR, SP, DIGIT, ALPHA and DQUOTE rules from [RFC5234]. It also includes the tchar and OWS rules from [RFC7230].

When parsing from HTTP fields, implementations MUST have behavior that is indistinguishable from following the algorithms. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence.

For serialization to HTTP fields, the ABNF illustrates their expected wire representations, and the algorithms define the recommended way to produce them. Implementations MAY vary from the specified behavior so long as the output is still correctly handled by the parsing algorithm.

## 2. Defining New Structured Fields

To specify a HTTP field as a Structured Field, its authors needs to:

- o Normatively reference this specification. Recipients and generators of the field need to know that the requirements of this document are in effect.

- o Identify whether the field is a Structured Header (i.e., it can only be used in the header section - the common case), a Structured Trailer (only in the trailer section), or a Structured Field (both).
- o Specify the type of the field value; either List (Section 3.1), Dictionary (Section 3.2), or Item (Section 3.3).
- o Define the semantics of the field value.
- o Specify any additional constraints upon the field value, as well as the consequences when those constraints are violated.

Typically, this means that a field definition will specify the top-level type - List, Dictionary or Item - and then define its allowable types, and constraints upon them. For example, a header defined as a List might have all Integer members, or a mix of types; a header defined as an Item might allow only Strings, and additionally only strings beginning with the letter "Q", or strings in lowercase. Likewise, Inner Lists (Section 3.1.1) are only valid when a field definition explicitly allows them.

When parsing fails, the entire field is ignored (see Section 4.2); in most situations, violating field-specific constraints should have the same effect. Thus, if a header is defined as an Item and required to be an Integer, but a String is received, the field will by default be ignored. If the field requires different error handling, this should be explicitly specified.

Both Items and Inner Lists allow parameters as an extensibility mechanism; this means that values can later be extended to accommodate more information, if need be. To preserve forward compatibility, field specifications are discouraged from defining the presence of an unrecognized Parameter as an error condition.

To further assure that this extensibility is available in the future, and to encourage consumers to use a complete parser implementation, a field definition can specify that "grease" Parameters be added by senders. A specification could stipulate that all Parameters that fit a defined pattern are reserved for this use and then encourage them to be sent on some portion of requests. This helps to discourage recipients from writing a parser that does not account for Parameters.

Specifications that use Dictionaries can also allow for forward compatibility by requiring that the presence of - as well as value and type associated with - unknown members be ignored. Later

specifications can then add additional members, specifying constraints on them as appropriate.

An extension to a structured field can then require that an entire field value be ignored by a recipient that understands the extension if constraints on the value it defines are not met.

A field definition cannot relax the requirements of this specification because doing so would preclude handling by generic software; they can only add additional constraints (for example, on the numeric range of Integers and Decimals, the format of Strings and Tokens, the types allowed in a Dictionary's values, or the number of Items in a List). Likewise, field definitions can only use this specification for the entire field value, not a portion thereof.

This specification defines minimums for the length or number of various structures supported by implementations. It does not specify maximum sizes in most cases, but authors should be aware that HTTP implementations do impose various limits on the size of individual fields, the total number of fields, and/or the size of the entire header or trailer section.

Specifications can refer to a field name as a "structured header name", "structured trailer name" or "structured field name" as appropriate. Likewise, they can refer its field value as a "structured header value", "structured trailer value" or "structured field value" as necessary. Field definitions are encouraged to use the ABNF rules beginning with "sf-" defined in this specification; other rules in this specification are not intended for their use.

For example, a fictitious Foo-Example header field might be specified as:

--8<--

#### 42. Foo-Example Header

The Foo-Example HTTP header field conveys information about how much Foo the message has.

Foo-Example is a Item Structured Header [RFCxxxx]. Its value MUST be an Integer (Section Y.Y of [RFCxxxx]). Its ABNF is:

```
Foo-Example = sf-integer
```

Its value indicates the amount of Foo in the message, and MUST be between 0 and 10, inclusive; other values MUST cause the entire header field to be ignored.

The following parameters are defined:

- \* A Parameter whose name is "foourl", and whose value is a String (Section Y.Y of [RFCxxxx]), conveying the Foo URL for the message. See below for processing requirements.

"foourl" contains a URI-reference (Section 4.1 of [RFC3986]). If its value is not a valid URI-reference, the entire header field MUST be ignored. If its value is a relative reference (Section 4.2 of [RFC3986]), it MUST be resolved (Section 5 of [RFC3986]) before being used.

For example:

```
Foo-Example: 2; foourl="https://foo.example.com/"
-->8--
```

### 3. Structured Data Types

This section defines the abstract types for Structured Fields. The ABNF provided represents the on-wire format in HTTP field values.

In summary:

- o There are three top-level types that a HTTP field can be defined as: Lists, Dictionaries, and Items.
- o Lists and Dictionaries are containers; their members can be Items or Inner Lists (which are themselves arrays of Items).
- o Both Items and Inner Lists can be parameterized with key/value pairs.

### 3.1. Lists

Lists are arrays of zero or more members, each of which can be an Item (Section 3.3) or an Inner List (Section 3.1.1), both of which can be Parameterized (Section 3.1.2).

The ABNF for Lists in HTTP fields is:

```
sf-list      = list-member *( OWS "," OWS list-member )
list-member  = sf-item / inner-list
```

Each member is separated by a comma and optional whitespace. For example, a field whose value is defined as a List of Strings could look like:

```
Example-StrList: "foo", "bar", "It was the best of times."
```

An empty List is denoted by not serializing the field at all. This implies that fields defined as Lists have a default empty value.

Note that Lists can have their members split across multiple lines inside a header or trailer section, as per Section 3.2.2 of [RFC7230]; for example, the following are equivalent:

```
Example-Hdr: foo, bar
```

and

```
Example-Hdr: foo
Example-Hdr: bar
```

However, individual members of a List cannot be safely split between across lines; see Section 4.2 for details.

Parsers MUST support Lists containing at least 1024 members. Field specifications can constrain the types and cardinality of individual List values as they require.

#### 3.1.1. Inner Lists

An Inner List is an array of zero or more Items (Section 3.3). Both the individual Items and the Inner List itself can be Parameterized (Section 3.1.2).

The ABNF for Inner Lists is:

```
inner-list   = "(" *SP [ sf-item *( 1*SP sf-item ) *SP ] ")"
              parameters
```



Inner Lists are denoted by surrounding parenthesis, and have their values delimited by one or more spaces. A field whose value is defined as a List of Inner Lists of Strings could look like:

Example-StrListList: ("foo" "bar"), ("baz"), ("bat" "one"), ()

Note that the last member in this example is an empty Inner List.

A header field whose value is defined as a List of Inner Lists with Parameters at both levels could look like:

Example-ListListParam: ("foo"; a=1;b=2);lvl=5, ("bar" "baz");lvl=1

Parsers MUST support Inner Lists containing at least 256 members. Field specifications can constrain the types and cardinality of individual Inner List members as they require.

### 3.1.2. Parameters

Parameters are an ordered map of key-value pairs that are associated with an Item (Section 3.3) or Inner List (Section 3.1.1). The keys are unique within the scope the Parameters they occur within, and the values are bare items (i.e., they themselves cannot be parameterized; see Section 3.3).

The ABNF for Parameters is:

```
parameters    = *( ";" *SP parameter )
parameter     = param-name [ "=" param-value ]
param-name    = key
key           = ( lcalpha / "*" )
              *( lcalpha / DIGIT / "_" / "-" / "." / "*" )
lcalpha       = %x61-7A ; a-z
param-value   = bare-item
```

Note that Parameters are ordered as serialized, and Parameter keys cannot contain uppercase letters. A parameter is separated from its Item or Inner List and other parameters by a semicolon. For example:

Example-ParamList: abc;a=1;b=2; cde\_456, (ghi;jk=4 l);q="9";r=w

Parameters whose value is Boolean (see Section 3.3.6) true MUST omit that value when serialized. For example, the "a" parameter here is true, while the "b" parameter is false:

Example-Int: 1; a; b=?0

Note that this requirement is only on serialization; parsers are still required to correctly handle the true value when it appears in a parameter.

Parsers MUST support at least 256 parameters on an Item or Inner List, and support parameter keys with at least 64 characters. Field specifications can constrain the order of individual Parameters, as well as their values' types as required.

### 3.2. Dictionaries

Dictionaries are ordered maps of name-value pairs, where the names are short textual strings and the values are Items (Section 3.3) or arrays of Items, both of which can be Parameterized (Section 3.1.2). There can be zero or more members, and their names are unique in the scope of the Dictionary they occur within.

Implementations MUST provide access to Dictionaries both by index and by name. Specifications MAY use either means of accessing the members.

The ABNF for Dictionaries is:

```
sf-dictionary = dict-member *( OWS "," OWS dict-member )
dict-member  = member-name [ "=" member-value ]
member-name   = key
member-value  = sf-item / inner-list
```

Members are ordered as serialized, and separated by a comma with optional whitespace. Member names cannot contain uppercase characters. Names and values are separated by "=" (without whitespace). For example:

Example-Dict: en="Applepie", da=:w4ZibGV0w6ZydGU=:

Note that in this example, the final "=" is due to the inclusion of a Byte Sequence; see Section 3.3.5.

Members whose value is Boolean (see Section 3.3.6) true MUST omit that value when serialized. For example, here both "b" and "c" are true:

Example-Dict: a=?0, b, c; foo=bar

Note that this requirement is only on serialization; parsers are still required to correctly handle the true Boolean value when it appears in Dictionary values.

A Dictionary with a member whose value is an Inner List of Tokens:

Example-DictList: rating=1.5, feelings=(joy sadness)

A Dictionary with a mix of Items and Inner Lists, some with Parameters:

Example-MixDict: a=(1 2), b=3, c=4;aa=bb, d=(5 6);valid

As with lists, an empty Dictionary is represented by omitting the entire field. This implies that fields defined as Dictionaries have a default empty value.

Typically, a field specification will define the semantics of Dictionaries by specifying the allowed type(s) for individual members by their names, as well as whether their presence is required or optional. Recipients MUST ignore names that are undefined or unknown, unless the field's specification specifically disallows them.

Note that Dictionaries can have their members split across multiple lines inside a header or trailer section; for example, the following are equivalent:

Example-Hdr: foo=1, bar=2

and

Example-Hdr: foo=1

Example-Hdr: bar=2

However, individual members of a Dictionary cannot be safely split between lines; see Section 4.2 for details.

Parsers MUST support Dictionaries containing at least 1024 name/value pairs, and names with at least 64 characters. Field specifications can constrain the order of individual Dictionary members, as well as their values' types as required.

### 3.3. Items

An Item can be a Integer (Section 3.3.1), Decimal (Section 3.3.2), String (Section 3.3.3), Token (Section 3.3.4), Byte Sequence (Section 3.3.5), or Boolean (Section 3.3.6). It can have associated Parameters (Section 3.1.2).

The ABNF for Items is:

```
sf-item    = bare-item parameters
bare-item  = sf-integer / sf-decimal / sf-string / sf-token
           / sf-binary / sf-boolean
```

For example, a header field that is defined to be an Item that is an Integer might look like:

Example-IntItemHeader: 5

or with Parameters:

Example-IntItem: 5; foo=bar

### 3.3.1. Integers

Integers have a range of -999,999,999,999,999 to 999,999,999,999,999 inclusive (i.e., up to fifteen digits, signed), for IEEE 754 compatibility ([IEEE754]).

The ABNF for Integers is:

```
sf-integer = ["-"] 1*15DIGIT
```

For example:

Example-Integer: 42

Integers larger than 15 digits can be supported in a variety of ways; for example, by using a String (Section 3.3.3), Byte Sequence (Section 3.3.5), or a parameter on an Integer that acts as a scaling factor.

While it is possible to serialise Integers with leading zeros (e.g., "0002", "-01") and signed zero ("-0"), these distinctions may not be preserved by implementations.

Note that commas in Integers are used in this section's prose only for readability; they are not valid in the wire format.

### 3.3.2. Decimals

Decimals are numbers with an integer and a fractional component. The integer component has at most 12 digits; the fractional component has at most three digits.

The ABNF for decimals is:

```
sf-decimal = ["-"] 1*12DIGIT "." 1*3DIGIT
```

For example, a header whose value is defined as a Decimal could look like:

Example-Decimal: 4.5

While it is possible to serialise Decimals with leading zeros (e.g., "0002.5", "-01.334"), trailing zeros (e.g., "5.230", "-0.40"), and signed zero (e.g., "-0.0"), these distinctions may not be preserved by implementations.

Note that the serialisation algorithm (Section 4.1.5) rounds input with more than three digits of precision in the fractional component. If an alternative rounding strategy is desired, this should be specified by the header definition to occur before serialisation.

### 3.3.3. Strings

Strings are zero or more printable ASCII [RFC0020] characters (i.e., the range %x20 to %x7E). Note that this excludes tabs, newlines, carriage returns, etc.

The ABNF for Strings is:

```
sf-string = DQUOTE *chr DQUOTE
chr       = unescaped / escaped
unescaped = %x20-21 / %x23-5B / %x5D-7E
escaped   = "\" ( DQUOTE / "\" )
```

Strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

Example-String: "hello world"

Note that Strings only use DQUOTE as a delimiter; single quotes do not delimit Strings. Furthermore, only DQUOTE and "\"" can be escaped; other characters after "\"" MUST cause parsing to fail.

Unicode is not directly supported in Strings, because it causes a number of interoperability issues, and - with few exceptions - field values do not require it.

When it is necessary for a field value to convey non-ASCII content, a Byte Sequence (Section 3.3.5) can be specified, along with a character encoding (preferably [UTF-8]).

Parsers MUST support Strings (after any decoding) with at least 1024 characters.

### 3.3.4. Tokens

Tokens are short textual words; their abstract model is identical to their expression in the HTTP field value serialization.

The ABNF for Tokens is:

```
sf-token = ( ALPHA / "*" ) *( tchar / ":" / "/" )
```

For example:

Example-Token: foo123/456

Parsers MUST support Tokens with at least 512 characters.

Note that Token allows the same characters as the "token" ABNF rule defined in [RFC7230], with the exceptions that the first character is required to be either ALPHA or "\*", and ":" and "/" are also allowed in subsequent characters.

### 3.3.5. Byte Sequences

Byte Sequences can be conveyed in Structured Fields.

The ABNF for a Byte Sequence is:

```
sf-binary = ":" * (base64) ":"
base64    = ALPHA / DIGIT / "+" / "/" / "="
```

A Byte Sequence is delimited with colons and encoded using base64 ([RFC4648], Section 4). For example:

Example-Binary: :cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==:

Parsers MUST support Byte Sequences with at least 16384 octets after decoding.

### 3.3.6. Booleans

Boolean values can be conveyed in Structured Fields.

The ABNF for a Boolean is:

```
sf-boolean = "?" boolean
boolean    = "0" / "1"
```

A Boolean is indicated with a leading "?" character followed by a "1" for a true value or "0" for false. For example:

Example-Bool: ?1

Note that in Dictionary (Section 3.2) and Parameter (Section 3.1.2) values, Boolean true is indicated by omitting the value.

#### 4. Working With Structured Fields in HTTP

This section defines how to serialize and parse Structured Fields in textual HTTP field values and other encodings compatible with them (e.g., in HTTP/2 [RFC7540] before compression with HPACK [RFC7541]).

##### 4.1. Serializing Structured Fields

Given a structure defined in this specification, return an ASCII string suitable for use in a HTTP field value.

1. If the structure is a Dictionary or List and its value is empty (i.e., it has no members), do not serialize the field at all (i.e., omit both the field-name and field-value).
2. If the structure is a List, let `output_string` be the result of running Serializing a List (Section 4.1.1) with the structure.
3. Else if the structure is a Dictionary, let `output_string` be the result of running Serializing a Dictionary (Section 4.1.2) with the structure.
4. Else if the structure is an Item, let `output_string` be the result of running Serializing an Item (Section 4.1.3) with the structure.
5. Else, fail serialization.
6. Return `output_string` converted into an array of bytes, using ASCII encoding [RFC0020].

###### 4.1.1. Serializing a List

Given an array of (member\_value, parameters) tuples as `input_list`, return an ASCII string suitable for use in a HTTP field value.

1. Let `output` be an empty string.
2. For each (member\_value, parameters) of `input_list`:
  1. If `member_value` is an array, append the result of running Serializing an Inner List (Section 4.1.1.1) with (member\_value, parameters) to `output`.

2. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member\_value, parameters) to output.
3. If more member\_values remain in input\_list:
  1. Append ",", to output.
  2. Append a single SP to output.
3. Return output.

#### 4.1.1.1. Serializing an Inner List

Given an array of (member\_value, parameters) tuples as inner\_list, and parameters as list\_parameters, return an ASCII string suitable for use in a HTTP field value.

1. Let output be the string "(".
2. For each (member\_value, parameters) of inner\_list:
  1. Append the result of running Serializing an Item (Section 4.1.3) with (member\_value, parameters) to output.
  2. If more values remain in inner\_list, append a single SP to output.
3. Append ")" to output.
4. Append the result of running Serializing Parameters (Section 4.1.1.2) with list\_parameters to output.
5. Return output.

#### 4.1.1.2. Serializing Parameters

Given an ordered Dictionary as input\_parameters (each member having a param\_name and a param\_value), return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.
2. For each param\_name with a value of param\_value in input\_parameters:
  1. Append ";" to output.



2. Append the result of running Serializing a Key (Section 4.1.1.3) with param\_name to output.
3. If param\_value is not Boolean true:
  1. Append "=" to output.
  2. Append the result of running Serializing a bare Item (Section 4.1.3.1) with param\_value to output.
3. Return output.

#### 4.1.1.3. Serializing a Key

Given a key as input\_key, return an ASCII string suitable for use in a HTTP field value.

1. Convert input\_key into a sequence of ASCII characters; if conversion fails, fail serialization.
2. If input\_key contains characters not in lcalpha, DIGIT, "\_", "-", ".", or "\*" fail serialization.
3. If the first character of input\_key is not lcalpha or "\*", fail serialization.
4. Let output be an empty string.
5. Append input\_key to output.
6. Return output.

#### 4.1.2. Serializing a Dictionary

Given an ordered Dictionary as input\_dictionary (each member having a member\_name and a tuple value of (member\_value, parameters)), return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.
2. For each member\_name with a value of (member\_value, parameters) in input\_dictionary:
  1. Append the result of running Serializing a Key (Section 4.1.1.3) with member's member\_name to output.
  2. If member\_value is Boolean true:

1. Append the result of running Serializing Parameters (Section 4.1.1.2) with parameters to output.
3. Otherwise:
  1. Append "=" to output.
  2. If member\_value is an array, append the result of running Serializing an Inner List (Section 4.1.1.1) with (member\_value, parameters) to output.
  3. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member\_value, parameters) to output.
4. If more members remain in input\_dictionary:
  1. Append "," to output.
  2. Append a single SP to output.
3. Return output.

#### 4.1.3. Serializing an Item

Given an Item as bare\_item and Parameters as item\_parameters, return an ASCII string suitable for use in a HTTP field value.

1. Let output be an empty string.
2. Append the result of running Serializing a Bare Item Section 4.1.3.1 with bare\_item to output.
3. Append the result of running Serializing Parameters Section 4.1.1.2 with item\_parameters to output.
4. Return output.

##### 4.1.3.1. Serializing a Bare Item

Given an Item as input\_item, return an ASCII string suitable for use in a HTTP field value.

1. If input\_item is an Integer, return the result of running Serializing an Integer (Section 4.1.4) with input\_item.
2. If input\_item is a Decimal, return the result of running Serializing a Decimal (Section 4.1.5) with input\_item.

3. If `input_item` is a String, return the result of running Serializing a String (Section 4.1.6) with `input_item`.
4. If `input_item` is a Token, return the result of running Serializing a Token (Section 4.1.7) with `input_item`.
5. If `input_item` is a Boolean, return the result of running Serializing a Boolean (Section 4.1.9) with `input_item`.
6. If `input_item` is a Byte Sequence, return the result of running Serializing a Byte Sequence (Section 4.1.8) with `input_item`.
7. Otherwise, fail serialization.

#### 4.1.4. Serializing an Integer

Given an Integer as `input_integer`, return an ASCII string suitable for use in a HTTP field value.

1. If `input_integer` is not an integer in the range of -999,999,999,999,999 to 999,999,999,999,999 inclusive, fail serialization.
2. Let `output` be an empty string.
3. If `input_integer` is less than (but not equal to) 0, append "-" to `output`.
4. Append `input_integer`'s numeric value represented in base 10 using only decimal digits to `output`.
5. Return `output`.

#### 4.1.5. Serializing a Decimal

Given a decimal number as `input_decimal`, return an ASCII string suitable for use in a HTTP field value.

1. If `input_decimal` is not a decimal number, fail serialization.
2. If `input_decimal` has more than three significant digits to the right of the decimal point, round it to three decimal places, rounding the final digit to the nearest value, or to the even value if it is equidistant.
3. If `input_decimal` has more than 12 significant digits to the left of the decimal point after rounding, fail serialization.

4. Let output be an empty string.
5. If input\_decimal is less than (but not equal to) 0, append "-" to output.
6. Append input\_decimal's integer component represented in base 10 (using only decimal digits) to output; if it is zero, append "0".
7. Append "." to output.
8. If input\_decimal's fractional component is zero, append "0" to output.
9. Otherwise, append the significant digits of input\_decimal's fractional component represented in base 10 (using only decimal digits) to output.
10. Return output.

#### 4.1.6. Serializing a String

Given a String as input\_string, return an ASCII string suitable for use in a HTTP field value.

1. Convert input\_string into a sequence of ASCII characters; if conversion fails, fail serialization.
2. If input\_string contains characters in the range %x00-1f or %x7f (i.e., not in VCHAR or SP), fail serialization.
3. Let output be the string DQUOTE.
4. For each character char in input\_string:
  1. If char is "\" or DQUOTE:
    1. Append "\" to output.
  2. Append char to output.
5. Append DQUOTE to output.
6. Return output.

#### 4.1.7. Serializing a Token

Given a Token as `input_token`, return an ASCII string suitable for use in a HTTP field value.

1. Convert `input_token` into a sequence of ASCII characters; if conversion fails, fail serialization.
2. If the first character of `input_token` is not ALPHA or "\*", or the remaining portion contains a character not in `tchar`, ":" or "/", fail serialization.
3. Let `output` be an empty string.
4. Append `input_token` to `output`.
5. Return `output`.

#### 4.1.8. Serializing a Byte Sequence

Given a Byte Sequence as `input_bytes`, return an ASCII string suitable for use in a HTTP field value.

1. If `input_bytes` is not a sequence of bytes, fail serialization.
2. Let `output` be an empty string.
3. Append ":" to `output`.
4. Append the result of base64-encoding `input_bytes` as per [RFC4648], Section 4, taking account of the requirements below.
5. Append ":" to `output`.
6. Return `output`.

The encoded data is required to be padded with "=", as per [RFC4648], Section 3.2.

Likewise, encoded data SHOULD have pad bits set to zero, as per [RFC4648], Section 3.5, unless it is not possible to do so due to implementation constraints.

#### 4.1.9. Serializing a Boolean

Given a Boolean as `input_boolean`, return an ASCII string suitable for use in a HTTP field value.

1. If `input_boolean` is not a boolean, fail serialization.
2. Let `output` be an empty string.
3. Append "?" to `output`.
4. If `input_boolean` is true, append "1" to `output`.
5. If `input_boolean` is false, append "0" to `output`.
6. Return `output`.

#### 4.2. Parsing Structured Fields

When a receiving implementation parses HTTP fields that are known to be Structured Fields, it is important that care be taken, as there are a number of edge cases that can cause interoperability or even security problems. This section specifies the algorithm for doing so.

Given an array of bytes `input_bytes` that represents the chosen field's field-value (which is empty if that field is not present), and `field_type` (one of "dictionary", "list", or "item"), return the parsed header value.

1. Convert `input_bytes` into an ASCII string `input_string`; if conversion fails, fail parsing.
2. Discard any leading SP characters from `input_string`.
3. If `field_type` is "list", let `output` be the result of running Parsing a List (Section 4.2.1) with `input_string`.
4. If `field_type` is "dictionary", let `output` be the result of running Parsing a Dictionary (Section 4.2.2) with `input_string`.
5. If `field_type` is "item", let `output` be the result of running Parsing an Item (Section 4.2.3) with `input_string`.
6. Discard any leading SP characters from `input_string`.
7. If `input_string` is not empty, fail parsing.
8. Otherwise, return `output`.

When generating `input_bytes`, parsers MUST combine all field lines in the same section (header or trailer) that case-insensitively match the field name into one comma-separated field-value, as per

[RFC7230], Section 3.2.2; this assures that the entire field value is processed correctly.

For Lists and Dictionaries, this has the effect of correctly concatenating all of the field's lines, as long as individual members of the top-level data structure are not split across multiple header instances. The parsing algorithms for both types allow tab characters, since these might be used to combine field lines by some implementations.

Strings split across multiple field lines will have unpredictable results, because comma(s) and whitespace inserted upon combination will become part of the string output by the parser. Since concatenation might be done by an upstream intermediary, the results are not under the control of the serializer or the parser, even when they are both under the control of the same party.

Tokens, Integers, Decimals and Byte Sequences cannot be split across multiple field lines because the inserted commas will cause parsing to fail.

Parsers MAY fail when processing a field value spread across multiple field lines, when one of those lines does not parse as that field. For example, a parsing handling an Example-String field that's defined as a sf-string is allowed to fail when processing this field section:

```
Example-String: "foo
Example-String: bar"
```

If parsing fails - including when calling another algorithm - the entire field value MUST be ignored (i.e., treated as if the field were not present in the section). This is intentionally strict, to improve interoperability and safety, and specifications referencing this document are not allowed to loosen this requirement.

Note that this requirement does not apply to an implementation that is not parsing the field; for example, an intermediary is not required to strip a failing field from a message before forwarding it.

#### 4.2.1. Parsing a List

Given an ASCII string as `input_string`, return an array of (`item_or_inner_list`, `parameters`) tuples. `input_string` is modified to remove the parsed value.

1. Let `members` be an empty array.

2. While `input_string` is not empty:
  1. Append the result of running Parsing an Item or Inner List (Section 4.2.1.1) with `input_string` to `members`.
  2. Discard any leading OWS characters from `input_string`.
  3. If `input_string` is empty, return `members`.
  4. Consume the first character of `input_string`; if it is not `"`, fail parsing.
  5. Discard any leading OWS characters from `input_string`.
  6. If `input_string` is empty, there is a trailing comma; fail parsing.
3. No structured data has been found; return `members` (which is empty).

#### 4.2.1.1. Parsing an Item or Inner List

Given an ASCII string as `input_string`, return the tuple (`item_or_inner_list`, `parameters`), where `item_or_inner_list` can be either a single bare item, or an array of (`bare_item`, `parameters`) tuples. `input_string` is modified to remove the parsed value.

1. If the first character of `input_string` is `"`, return the result of running Parsing an Inner List (Section 4.2.1.2) with `input_string`.
2. Return the result of running Parsing an Item (Section 4.2.3) with `input_string`.

#### 4.2.1.2. Parsing an Inner List

Given an ASCII string as `input_string`, return the tuple (`inner_list`, `parameters`), where `inner_list` is an array of (`bare_item`, `parameters`) tuples. `input_string` is modified to remove the parsed value.

1. Consume the first character of `input_string`; if it is not `"`, fail parsing.
2. Let `inner_list` be an empty array.
3. While `input_string` is not empty:
  1. Discard any leading SP characters from `input_string`.



2. If the first character of `input_string` is `"")`:
    1. Consume the first character of `input_string`.
    2. Let `parameters` be the result of running Parsing Parameters (Section 4.2.3.2) with `input_string`.
    3. Return the tuple (`inner_list`, `parameters`).
  3. Let `item` be the result of running Parsing an Item (Section 4.2.3) with `input_string`.
  4. Append `item` to `inner_list`.
  5. If the first character of `input_string` is not SP or `"")`, fail parsing.
4. The end of the inner list was not found; fail parsing.

#### 4.2.2. Parsing a Dictionary

Given an ASCII string as `input_string`, return an ordered map whose values are (`item_or_inner_list`, `parameters`) tuples. `input_string` is modified to remove the parsed value.

1. Let `dictionary` be an empty, ordered map.
2. While `input_string` is not empty:
  1. Let `this_key` be the result of running Parsing a Key (Section 4.2.3.3) with `input_string`.
  2. If the first character of `input_string` is `"="`:
    1. Consume the first character of `input_string`.
    2. Let `member` be the result of running Parsing an Item or Inner List (Section 4.2.1.1) with `input_string`.
  3. Otherwise:
    1. Let `value` be Boolean true.
    2. Let `parameters` be the result of running Parsing Parameters Section 4.2.3.2 with `input_string`.
    3. Let `member` be the tuple (`value`, `parameters`).

4. Add name `this_key` with value member to dictionary. If dictionary already contains a name `this_key` (comparing character-for-character), overwrite its value.
  5. Discard any leading OWS characters from `input_string`.
  6. If `input_string` is empty, return dictionary.
  7. Consume the first character of `input_string`; if it is not `",`, fail parsing.
  8. Discard any leading OWS characters from `input_string`.
  9. If `input_string` is empty, there is a trailing comma; fail parsing.
3. No structured data has been found; return dictionary (which is empty).

Note that when duplicate Dictionary keys are encountered, this has the effect of ignoring all but the last instance.

#### 4.2.3. Parsing an Item

Given an ASCII string as `input_string`, return a (`bare_item`, `parameters`) tuple. `input_string` is modified to remove the parsed value.

1. Let `bare_item` be the result of running Parsing a Bare Item (Section 4.2.3.1) with `input_string`.
2. Let `parameters` be the result of running Parsing Parameters (Section 4.2.3.2) with `input_string`.
3. Return the tuple (`bare_item`, `parameters`).

##### 4.2.3.1. Parsing a Bare Item

Given an ASCII string as `input_string`, return a bare Item. `input_string` is modified to remove the parsed value.

1. If the first character of `input_string` is a `"-` or a DIGIT, return the result of running Parsing an Integer or Decimal (Section 4.2.4) with `input_string`.
2. If the first character of `input_string` is a DQUOTE, return the result of running Parsing a String (Section 4.2.5) with `input_string`.

3. If the first character of `input_string` is ":", return the result of running Parsing a Byte Sequence (Section 4.2.7) with `input_string`.
4. If the first character of `input_string` is "?", return the result of running Parsing a Boolean (Section 4.2.8) with `input_string`.
5. If the first character of `input_string` is an ALPHA or "\*", return the result of running Parsing a Token (Section 4.2.6) with `input_string`.
6. Otherwise, the item type is unrecognized; fail parsing.

#### 4.2.3.2. Parsing Parameters

Given an ASCII string as `input_string`, return an ordered map whose values are bare Items. `input_string` is modified to remove the parsed value.

1. Let `parameters` be an empty, ordered map.
2. While `input_string` is not empty:
  1. If the first character of `input_string` is not ";", exit the loop.
  2. Consume a ";" character from the beginning of `input_string`.
  3. Discard any leading SP characters from `input_string`.
  4. let `param_name` be the result of running Parsing a Key (Section 4.2.3.3) with `input_string`.
  5. Let `param_value` be Boolean true.
  6. If the first character of `input_string` is "=:
    1. Consume the "=" character at the beginning of `input_string`.
    2. Let `param_value` be the result of running Parsing a Bare Item (Section 4.2.3.1) with `input_string`.
  7. Append key `param_name` with value `param_value` to `parameters`. If `parameters` already contains a name `param_name` (comparing character-for-character), overwrite its value.
3. Return `parameters`.

Note that when duplicate Parameter keys are encountered, this has the effect of ignoring all but the last instance.

#### 4.2.3.3. Parsing a Key

Given an ASCII string as `input_string`, return a key. `input_string` is modified to remove the parsed value.

1. If the first character of `input_string` is not `lcalpha` or `"*"`, fail parsing.
2. Let `output_string` be an empty string.
3. While `input_string` is not empty:
  1. If the first character of `input_string` is not one of `lcalpha`, `DIGIT`, `"_"`, `"-"`, `"."`, or `"*"`, return `output_string`.
  2. Let `char` be the result of consuming the first character of `input_string`.
  3. Append `char` to `output_string`.
4. Return `output_string`.

#### 4.2.4. Parsing an Integer or Decimal

Given an ASCII string as `input_string`, return an Integer or Decimal. `input_string` is modified to remove the parsed value.

NOTE: This algorithm parses both Integers (Section 3.3.1) and Decimals (Section 3.3.2), and returns the corresponding structure.

1. Let `type` be `"integer"`.
2. Let `sign` be 1.
3. Let `input_number` be an empty string.
4. If the first character of `input_string` is `"-"`, consume it and set `sign` to -1.
5. If `input_string` is empty, there is an empty integer; fail parsing.
6. If the first character of `input_string` is not a `DIGIT`, fail parsing.

7. While input\_string is not empty:
    1. Let char be the result of consuming the first character of input\_string.
    2. If char is a DIGIT, append it to input\_number.
    3. Else, if type is "integer" and char is ".":
      1. If input\_number contains more than 12 characters, fail parsing.
      2. Otherwise, append char to input\_number and set type to "decimal".
    4. Otherwise, prepend char to input\_string, and exit the loop.
    5. If type is "integer" and input\_number contains more than 15 characters, fail parsing.
    6. If type is "decimal" and input\_number contains more than 16 characters, fail parsing.
  8. If type is "integer":
    1. Parse input\_number as an integer and let output\_number be the product of the result and sign.
    2. If output\_number is outside the range -999,999,999,999,999 to 999,999,999,999,999 inclusive, fail parsing.
  9. Otherwise:
    1. If the final character of input\_number is ".", fail parsing.
    2. If the number of characters after "." in input\_number is greater than three, fail parsing.
    3. Parse input\_number as a decimal number and let output\_number be the product of the result and sign.
  10. Return output\_number.
- 4.2.5. Parsing a String

Given an ASCII string as input\_string, return an unquoted String.  
input\_string is modified to remove the parsed value.

1. Let `output_string` be an empty string.
2. If the first character of `input_string` is not `DQUOTE`, fail parsing.
3. Discard the first character of `input_string`.
4. While `input_string` is not empty:
  1. Let `char` be the result of consuming the first character of `input_string`.
  2. If `char` is a backslash (`"\"`):
    1. If `input_string` is now empty, fail parsing.
    2. Let `next_char` be the result of consuming the first character of `input_string`.
    3. If `next_char` is not `DQUOTE` or `"\"`, fail parsing.
    4. Append `next_char` to `output_string`.
  3. Else, if `char` is `DQUOTE`, return `output_string`.
  4. Else, if `char` is in the range `%x00-1f` or `%x7f` (i.e., is not in `VCHAR` or `SP`), fail parsing.
  5. Else, append `char` to `output_string`.
5. Reached the end of `input_string` without finding a closing `DQUOTE`; fail parsing.

#### 4.2.6. Parsing a Token

Given an ASCII string as `input_string`, return a Token. `input_string` is modified to remove the parsed value.

1. If the first character of `input_string` is not `ALPHA` or `"*"`, fail parsing.
2. Let `output_string` be an empty string.
3. While `input_string` is not empty:
  1. If the first character of `input_string` is not in `tchar`, `":"` or `"/"`, return `output_string`.

2. Let char be the result of consuming the first character of input\_string.
3. Append char to output\_string.
4. Return output\_string.

#### 4.2.7. Parsing a Byte Sequence

Given an ASCII string as input\_string, return a Byte Sequence. input\_string is modified to remove the parsed value.

1. If the first character of input\_string is not ":", fail parsing.
2. Discard the first character of input\_string.
3. If there is not a ":" character before the end of input\_string, fail parsing.
4. Let b64\_content be the result of consuming content of input\_string up to but not including the first instance of the character ":".
5. Consume the ":" character at the beginning of input\_string.
6. If b64\_content contains a character not included in ALPHA, DIGIT, "+", "/" and "=", fail parsing.
7. Let binary\_content be the result of Base 64 Decoding [RFC4648] b64\_content, synthesizing padding if necessary (note the requirements about recipient behavior below).
8. Return binary\_content.

Because some implementations of base64 do not allow rejection of encoded data that is not properly "=" padded (see [RFC4648], Section 3.2), parsers SHOULD NOT fail when "=" padding is not present, unless they cannot be configured to do so.

Because some implementations of base64 do not allow rejection of encoded data that has non-zero pad bits (see [RFC4648], Section 3.5), parsers SHOULD NOT fail when non-zero pad bits are present, unless they cannot be configured to do so.

This specification does not relax the requirements in [RFC4648], Section 3.1 and 3.3; therefore, parsers MUST fail on characters outside the base64 alphabet, and on line feeds in encoded data.

#### 4.2.8. Parsing a Boolean

Given an ASCII string as `input_string`, return a Boolean. `input_string` is modified to remove the parsed value.

1. If the first character of `input_string` is not "?", fail parsing.
2. Discard the first character of `input_string`.
3. If the first character of `input_string` matches "1", discard the first character, and return true.
4. If the first character of `input_string` matches "0", discard the first character, and return false.
5. No value has matched; fail parsing.

#### 5. IANA Considerations

This document has no actions for IANA.

#### 6. Security Considerations

The size of most types defined by Structured Fields is not limited; as a result, extremely large fields could be an attack vector (e.g., for resource consumption). Most HTTP implementations limit the sizes of individual fields as well as the overall header or trailer section size to mitigate such attacks.

It is possible for parties with the ability to inject new HTTP fields to change the meaning of a Structured Field. In some circumstances, this will cause parsing to fail, but it is not possible to reliably fail in all such circumstances.

#### 7. References

##### 7.1. Normative References

- [RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <<https://www.rfc-editor.org/info/rfc20>>.
- [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/rfc2119>>.



- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [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/rfc5234>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [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/rfc8174>>.

## 7.2. Informative References

- [IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2019, DOI 10.1109/IEEESTD.2019.8766229, ISBN 978-1-5044-5924-2, July 2019, <<https://ieeexplore.ieee.org/document/8766229>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<https://www.rfc-editor.org/info/rfc7493>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<https://www.rfc-editor.org/info/rfc7541>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

[UTF-8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<http://www.rfc-editor.org/info/std63>>.

### 7.3. URIs

- [1] <https://lists.w3.org/Archives/Public/ietf-http-wg/>
- [2] <https://httpwg.github.io/>
- [3] <https://github.com/httpwg/http-extensions/labels/header-structure>
- [4] <https://github.com/httpwg/structured-field-tests>
- [5] <https://github.com/httpwg/wiki/wiki/Structured-Headers>
- [6] <https://github.com/httpwg/structured-field-tests>

## Appendix A. Frequently Asked Questions

### A.1. Why not JSON?

Earlier proposals for Structured Fields were based upon JSON [RFC8259]. However, constraining its use to make it suitable for HTTP header fields required senders and recipients to implement specific additional handling.

For example, JSON has specification issues around large numbers and objects with duplicate members. Although advice for avoiding these issues is available (e.g., [RFC7493]), it cannot be relied upon.

Likewise, JSON strings are by default Unicode strings, which have a number of potential interoperability issues (e.g., in comparison). Although implementers can be advised to avoid non-ASCII content where unnecessary, this is difficult to enforce.

Another example is JSON's ability to nest content to arbitrary depths. Since the resulting memory commitment might be unsuitable (e.g., in embedded and other limited server deployments), it's necessary to limit it in some fashion; however, existing JSON implementations have no such limits, and even if a limit is specified, it's likely that some field definition will find a need to violate it.

Because of JSON's broad adoption and implementation, it is difficult to impose such additional constraints across all implementations; some deployments would fail to enforce them, thereby harming

interoperability. In short, if it looks like JSON, people will be tempted to use a JSON parser / serializer on field values.

Since a major goal for Structured Fields is to improve interoperability and simplify implementation, these concerns led to a format that requires a dedicated parser and serializer.

Additionally, there were widely shared feelings that JSON doesn't "look right" in HTTP fields.

## Appendix B. Implementation Notes

A generic implementation of this specification should expose the top-level `serialize` (Section 4.1) and `parse` (Section 4.2) functions. They need not be functions; for example, it could be implemented as an object, with methods for each of the different top-level types.

For interoperability, it's important that generic implementations be complete and follow the algorithms closely; see Section 1.1. To aid this, a common test suite is being maintained by the community at <https://github.com/httpwg/structured-field-tests> [6].

Implementers should note that Dictionaries and Parameters are order-preserving maps. Some fields may not convey meaning in the ordering of these data types, but it should still be exposed so that applications which need to use it will have it available.

Likewise, implementations should note that it's important to preserve the distinction between Tokens and Strings. While most programming languages have native types that map to the other types well, it may be necessary to create a wrapper "token" object or use a parameter on functions to assure that these types remain separate.

The serialization algorithm is defined in a way that it is not strictly limited to the data types defined in Section 3 in every case. For example, Decimals are designed to take broader input and round to allowed values.

Implementations are allowed to limit the allowed size of different structures, subject to the minimums defined for each type. When a structure exceeds an implementation limit, that structure fails parsing or serialisation.

## Appendix C. Changes

\_\_RFC Editor: Please remove this section before publication.\_\_

## C.1. Since draft-ietf-httpbis-header-structure-18

- o Use "sf-" prefix for ABNF, not "sh-".
- o Fix indentation in Dictionary serialisation (#1164).
- o Add example for Token; tweak example field names (#1147).
- o Editorial improvements.
- o Note that exceeding implementation limits implies failure.
- o Talk about specifying order of Dictionary members and Parameters, not cardinality.
- o Allow (but don't require) parsers to fail when a single field line isn't valid.
- o Note that some aspects of Integers and Decimals are not necessarily preserved.
- o Allow Lists and Dictionaries to be delimited by OWS, rather than \*SP, to make parsing more robust.

## C.2. Since draft-ietf-httpbis-header-structure-17

- o Editorial improvements.

## C.3. Since draft-ietf-httpbis-header-structure-16

- o Editorial improvements.
- o Discussion on forwards compatibility.

## C.4. Since draft-ietf-httpbis-header-structure-15

- o Editorial improvements.
- o Use HTTP field terminology more consistently, in line with recent changes to HTTP-core.
- o String length requirements apply to decoded strings (#1051).
- o Correctly round decimals in serialisation (#1043).
- o Clarify input to serialisation algorithms (#1055).
- o Omitted True dictionary value can have parameters (#1083).

- o Keys can now start with '\*' (#1068).
- C.5. Since draft-ietf-httpbis-header-structure-14
- o Editorial improvements.
  - o Allow empty dictionary values (#992).
  - o Change value of omitted parameter value to True (#995).
  - o Explain more about splitting dictionaries and lists across header instances (#997).
  - o Disallow HTAB, replace OWS with spaces (#998).
  - o Change byte sequence delimiters from "\*" to ":" (#991).
  - o Allow tokens to start with "\*" (#991).
  - o Change Floats to fixed-precision Decimals (#982).
  - o Round the fractional component of decimal, rather than truncating it (#982).
  - o Handle duplicate dictionary and parameter keys by overwriting their values, rather than failing (#997).
  - o Allow "." in key (#1027).
  - o Check first character of key in serialisation (#1037).
  - o Talk about greasing headers (#1015).
- C.6. Since draft-ietf-httpbis-header-structure-13
- o Editorial improvements.
  - o Define "structured header name" and "structured header value" terms (#908).
  - o Corrected text about valid characters in strings (#931).
  - o Removed most instances of the word "textual", as it was redundant (#915).
  - o Allowed parameters on Items and Inner Lists (#907).
  - o Expand the range of characters in token (#961).

- o Disallow OWS before ";" delimiter in parameters (#961).
- C.7. Since draft-ietf-httpbis-header-structure-12
- o Editorial improvements.
  - o Reworked float serialisation (#896).
  - o Don't add a trailing space in inner-list (#904).
- C.8. Since draft-ietf-httpbis-header-structure-11
- o Allow \* in key (#844).
  - o Constrain floats to six digits of precision (#848).
  - o Allow dictionary members to have parameters (#842).
- C.9. Since draft-ietf-httpbis-header-structure-10
- o Update abstract (#799).
  - o Input and output are now arrays of bytes (#662).
  - o Implementations need to preserve difference between token and string (#790).
  - o Allow empty dictionaries and lists (#781).
  - o Change parameterized lists to have primary items (#797).
  - o Allow inner lists in both dictionaries and lists; removes lists of lists (#816).
  - o Subsume Parameterised Lists into Lists (#839).
- C.10. Since draft-ietf-httpbis-header-structure-09
- o Changed Boolean from T/F to 1/0 (#784).
  - o Parameters are now ordered maps (#765).
  - o Clamp integers to 15 digits (#737).

## C.11. Since draft-ietf-httpbis-header-structure-08

- o Disallow whitespace before items properly (#703).
- o Created "key" for use in dictionaries and parameters, rather than relying on identifier (#702). Identifiers have a separate minimum supported size.
- o Expanded the range of special characters allowed in identifier to include all of ALPHA, ".", ":", and "%" (#702).
- o Use "?" instead of "!" to indicate a Boolean (#719).
- o Added "Intentionally Strict Processing" (#684).
- o Gave better names for referring specs to use in Parameterised Lists (#720).
- o Added Lists of Lists (#721).
- o Rename Identifier to Token (#725).
- o Add implementation guidance (#727).

## C.12. Since draft-ietf-httpbis-header-structure-07

- o Make Dictionaries ordered mappings (#659).
- o Changed "binary content" to "byte sequence" to align with Infra specification (#671).
- o Changed "mapping" to "map" for #671.
- o Don't fail if byte sequences aren't "=" padded (#658).
- o Add Booleans (#683).
- o Allow identifiers in items again (#629).
- o Disallowed whitespace before items (#703).
- o Explain the consequences of splitting a string across multiple headers (#686).

- C.13. Since draft-ietf-httpbis-header-structure-06
- o Add a FAQ.
  - o Allow non-zero pad bits.
  - o Explicitly check for integers that violate constraints.
- C.14. Since draft-ietf-httpbis-header-structure-05
- o Reorganise specification to separate parsing out.
  - o Allow referencing specs to use ABNF.
  - o Define serialisation algorithms.
  - o Refine relationship between ABNF, parsing and serialisation algorithms.
- C.15. Since draft-ietf-httpbis-header-structure-04
- o Remove identifiers from item.
  - o Remove most limits on sizes.
  - o Refine number parsing.
- C.16. Since draft-ietf-httpbis-header-structure-03
- o Strengthen language around failure handling.
- C.17. Since draft-ietf-httpbis-header-structure-02
- o Split Numbers into Integers and Floats.
  - o Define number parsing.
  - o Tighten up binary parsing and give it an explicit end delimiter.
  - o Clarify that mappings are unordered.
  - o Allow zero-length strings.
  - o Improve string parsing algorithm.
  - o Improve limits in algorithms.
  - o Require parsers to combine header fields before processing.



- o Throw an error on trailing garbage.
- C.18. Since draft-ietf-httpbis-header-structure-01
- o Replaced with draft-nottingham-structured-headers.
- C.19. Since draft-ietf-httpbis-header-structure-00
- o Added signed 64bit integer type.
  - o Drop UTF8, and settle on BCP137 ::EmbeddedUnicodeChar for hl-unicode-string.
  - o Change hl\_blob delimiter to ":" since "'" is valid t\_char

#### Acknowledgements

Many thanks to Matthew Kerwin for his detailed feedback and careful consideration during the development of this specification.

Thanks also to Ian Clelland, Roy Fielding, Anne van Kesteren, Kazuho Oku, Evert Pot, Julian Reschke, Martin Thomson, Mike West, and Jeffrey Yasskin for their contributions.

#### Authors' Addresses

Mark Nottingham  
Fastly

Email: [mnot@mnot.net](mailto:mnot@mnot.net)  
URI: <https://www.mnot.net/>

Poul-Henning Kamp  
The Varnish Cache Project

Email: [phk@varnish-cache.org](mailto:phk@varnish-cache.org)

HTTP Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: January 4, 2018

P. McManus  
Mozilla  
July 3, 2017

HTTP Immutable Responses  
draft-ietf-httpbis-immutable-03

Abstract

The immutable HTTP response Cache-Control extension allows servers to identify resources that will not be updated during their freshness lifetime. This assures that a client never needs to revalidate a cached fresh resource to be certain it has not been modified.

Note to Readers

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

Working Group information can be found at <http://httpwg.github.io/>; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/immutable>.

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 4, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## 1. Introduction

HTTP's freshness lifetime mechanism [RFC7234] allows a client to safely reuse a stored response to satisfy future requests for a specified period of time. However, it is still possible that the resource will be modified during that period.

For instance, a front page newspaper photo with a freshness lifetime of one hour would mean that no user would see a cached photo more than one hour old. However, the photo could be updated at any time resulting in different users seeing different photos depending on the contents of their caches for up to one hour. This is compliant with the caching mechanism defined in [RFC7234].

Users that need to confirm there have been no updates to their cached responses typically use the reload (or refresh) mechanism in their user agents. This in turn generates a conditional request [RFC7232] and either a new representation or, if unmodified, a 304 (Not Modified) response [RFC7232] is returned. A user agent that understands HTML and fetches its dependent sub-resources might issue hundreds of conditional requests to refresh all portions of a common page [REQPERPAGE].

However some content providers never create more than one variant of a sub-resource, because they use "versioned" URLs. When these resources need an update they are simply published under a new URL, typically embedding an identifier unique to that version of the resource in the path, and references to the sub-resource are updated with the new path information.

For example, "<https://www.example.com/101016/main.css>" might be updated and republished as "<https://www.example.com/102026/main.css>", with any links that reference it being changed at the same time. This design pattern allows a very large freshness lifetime to be used for the sub-resource without guessing when it will be updated in the future.

Unfortunately, the user agent does not know when this versioned URL design pattern is used. As a result, user-driven refreshes still translate into wasted conditional requests for each sub-resource as each will return 304 responses.

The "immutable" HTTP response Cache-Control extension allows servers to identify responses that will not be updated during their freshness lifetimes.

This effectively informs clients that any conditional request for that response can be safely skipped without worrying that it has been updated.

### 1.1. Notational Conventions

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].

## 2. The immutable Cache-Control extension

When present in an HTTP response, the "immutable" Cache-Control extension indicates that the origin server will not update the representation of that resource during the freshness lifetime of the response.

Clients SHOULD NOT issue a conditional request during the response's freshness lifetime (e.g. upon a reload) unless explicitly overridden by the user (e.g. a force reload).

The immutable extension only applies during the freshness lifetime of the stored response. Stale responses SHOULD be revalidated as they normally would be in the absence of immutable.

The immutable extension takes no arguments. If any arguments are present, they have no meaning, and MUST be ignored. Multiple instances of the immutable extension are equivalent to one instance. The presence of an immutable Cache-Control extension in a request has no effect.

### 2.1. About Intermediaries

An immutable response has the same semantic meaning when received by proxy clients as it does when received by User-Agent based clients. Therefore proxies SHOULD skip conditionally revalidating fresh responses containing the immutable extension unless there is a signal from the client that a validation is necessary (e.g. a no-cache

Cache-Control request directive defined by Section 5.2.1.4 of [RFC7234]).

A proxy that uses immutable to bypass a conditional revalidation can choose whether to reply with a 304 or 200 to its requesting client based on the request headers the proxy received.

## 2.2. Example

Cache-Control: max-age=31536000, immutable

## 3. Security Considerations

The immutable mechanism acts as form of soft pinning and, as with all pinning mechanisms, creates a vector for amplification of cache corruption incidents. These incidents include cache poisoning attacks. Three mechanisms are suggested for mitigation of this risk:

- o Clients SHOULD ignore immutable from resources that are not part of an authenticated context such as HTTPS. Authenticated resources are less vulnerable to cache poisoning.
- o User-Agents often provide two different refresh mechanisms: reload and some form of force-reload. The latter is used to rectify interrupted loads and other corruption. These reloads, typically indicated through no-cache request attributes, SHOULD ignore immutable as well.
- o Clients SHOULD ignore immutable for resources that do not provide a strong indication that the stored response size is the correct response size such as responses delimited by connection close.

## 4. IANA Considerations

Section 7.1 of [RFC7234] requires registration of the immutable extension in the "Hypertext Transfer Protocol (HTTP) Cache Directive Registry" with IETF Review.

- o Cache-Directive: immutable
- o Pointer to specification text: [this document]

## 5. Acknowledgments

Thank you to Ben Maurer for partnership in developing and testing this idea. Thank you to Amos Jeffries for help with proxy interactions and to Mark Nottingham for help with the documentation.

## 6. References

### 6.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <<http://www.rfc-editor.org/info/rfc7232>>.
- [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, <<http://www.rfc-editor.org/info/rfc7234>>.

### 6.2. Informative References

- [REQPERPAGE] "HTTP Archive", n.d., <<http://httparchive.org/interesting.php#reqTotal>>.

### Author's Address

Patrick McManus  
Mozilla

Email: [pmcmanus@mozilla.com](mailto:pmcmanus@mozilla.com)

HTTP  
Internet-Draft  
Intended status: Standards Track  
Expires: July 17, 2018

M. Nottingham  
E. Nygren  
Akamai  
January 13, 2018

The ORIGIN HTTP/2 Frame  
draft-ietf-httpbis-origin-frame-06

Abstract

This document specifies the ORIGIN frame for HTTP/2, to indicate what origins are available on a given connection.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list ([ietf-http-wg@w3.org](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> [1].

Working Group information can be found at <http://httpwg.github.io/> [2]; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/origin-frame> [3].

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 <https://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 July 17, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
1.1. Notational Conventions . . . . .	3
2. The ORIGIN HTTP/2 Frame . . . . .	3
2.1. Syntax . . . . .	3
2.2. Processing ORIGIN Frames . . . . .	4
2.3. The Origin Set . . . . .	5
2.4. Authority, Push and Coalescing with ORIGIN . . . . .	6
3. IANA Considerations . . . . .	7
4. Security Considerations . . . . .	7
5. References . . . . .	7
5.1. Normative References . . . . .	7
5.2. Informative References . . . . .	8
5.3. URIs . . . . .	9
Appendix A. Non-Normative Processing Algorithm . . . . .	9
Appendix B. Operational Considerations for Servers . . . . .	9
Authors' Addresses . . . . .	10

## 1. Introduction

HTTP/2 [RFC7540] allows clients to coalesce different origins [RFC6454] onto the same connection when certain conditions are met. However, in certain cases, a connection is not usable for a coalesced origin, so the 421 (Misdirected Request) status code ([RFC7540], Section 9.1.2) was defined.

Using a status code in this manner allows clients to recover from misdirected requests, but at the penalty of adding latency. To address that, this specification defines a new HTTP/2 frame type, "ORIGIN", to allow servers to indicate what origins a connection is usable for.

Additionally, experience has shown that HTTP/2's requirement to establish server authority using both DNS and the server's certificate is onerous. This specification relaxes the requirement to check DNS when the ORIGIN frame is in use. Doing so has



additional benefits, such as removing the latency associated with some DNS lookups.

### 1.1. Notational Conventions

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.

## 2. The ORIGIN HTTP/2 Frame

This document defines a new HTTP/2 frame type ([RFC7540], Section 4) called ORIGIN, that allows a server to indicate what origin(s) [RFC6454] the server would like the client to consider as members of the Origin Set (Section 2.3) for the connection it occurs within.

### 2.1. Syntax

The ORIGIN frame type is 0xc (decimal 12), and contains zero or more instances of the Origin-Entry field.

```
+-----+-----+
|          Origin-Entry (*)          ...
+-----+-----+
```

An Origin-Entry is a length-delimited string:

```
+-----+-----+
|          Origin-Len (16)          | ASCII-Origin?          ...
+-----+-----+
```

Specifically:

Origin-Len: An unsigned, 16-bit integer indicating the length, in octets, of the ASCII-Origin field.

Origin: An OPTIONAL sequence of characters containing the ASCII serialization of an origin ([RFC6454], Section 6.2) that the sender asserts this connection is or could be authoritative for.

The ORIGIN frame does not define any flags. However, future updates to this specification MAY define flags. See Section 2.2.

## 2.2. Processing ORIGIN Frames

The ORIGIN frame is a non-critical extension to HTTP/2. Endpoints that do not support this frame can safely ignore it upon receipt.

When received by an implementing client, it is used to initialise and manipulate the Origin Set (see Section 2.3), thereby changing how the client establishes authority for origin servers (see Section 2.4).

The ORIGIN frame MUST be sent on stream 0; an ORIGIN frame on any other stream is invalid and MUST be ignored.

Likewise, the ORIGIN frame is only valid on connections with the "h2" protocol identifier, or when specifically nominated by the protocol's definition; it MUST be ignored when received on a connection with the "h2c" protocol identifier.

This specification does not define any flags for the ORIGIN frame, but future updates to this specification (through IETF consensus) might use them to change its semantics. The first four flags (0x1, 0x2, 0x4 and 0x8) are reserved for backwards-incompatible changes, and therefore when any of them are set, the ORIGIN frame containing them MUST be ignored by clients conforming to this specification, unless the flag's semantics are understood. The remaining flags are reserved for backwards-compatible changes, and do not affect processing by clients conformant to this specification.

The ORIGIN frame describes a property of the connection, and therefore is processed hop-by-hop. An intermediary MUST NOT forward ORIGIN frames. Clients configured to use a proxy MUST ignore any ORIGIN frames received from it.

Each ASCII-Origin field in the frame's payload MUST be parsed as an ASCII serialisation of an origin ([RFC6454], Section 6.2). If parsing fails, the field MUST be ignored.

Note that the ORIGIN frame does not support wildcard names (e.g., "\*.example.com") in Origin-Entry. As a result, sending ORIGIN when a wildcard certificate is in use effectively disables any origins that are not explicitly listed in the ORIGIN frame(s) (when the client understands ORIGIN).

See Appendix A for an illustrative algorithm for processing ORIGIN frames.

### 2.3. The Origin Set

The set of origins (as per [RFC6454]) that a given connection might be used for is known in this specification as the Origin Set.

By default, the Origin Set for a connection is uninitialised. An uninitialized Origin Set means that clients apply the coalescing rules from Section 9.1.1 of [RFC7540].

When an ORIGIN frame is first received and successfully processed by a client, the connection's Origin Set is defined to contain an initial origin. The initial origin is composed from:

- o Scheme: "https"
- o Host: the value sent in Server Name Indication (SNI, [RFC6066], Section 3), converted to lower case; if SNI is not present, the remote address of the connection (i.e., the server's IP address)
- o Port: the remote port of the connection (i.e., the server's port)

The contents of that ORIGIN frame (and subsequent ones) allows the server to incrementally add new origins to the Origin Set, as described in Section 2.2.

The Origin Set is also affected by the 421 (Misdirected Request) response status code, defined in [RFC7540], Section 9.1.2. Upon receipt of a response with this status code, implementing clients MUST create the ASCII serialisation of the corresponding request's origin (as per [RFC6454], Section 6.2) and remove it from the connection's Origin Set, if present.

**Note:** When sending an ORIGIN frame to a connection that is initialised as an Alternative Service [RFC7838], the initial origin set (Section 2.3) will contain an origin with the appropriate scheme and hostname (since Alternative Services specifies that the origin's hostname be sent in SNI). However, it is possible that the port will be different than that of the intended origin, since the initial origin set is calculated using the actual port in use, which can be different for the alternative service. In this case, the intended origin needs to be sent in the ORIGIN frame explicitly.

For example, a client making requests for "https://example.com" is directed to an alternative service at ("h2", "x.example.net", "8443"). If this alternative service sends an ORIGIN frame, the initial origin will be "https://example.com:8443". The client will not be able to use the alternative service to make requests

for "https://example.com" unless that origin is explicitly included in the ORIGIN frame.

#### 2.4. Authority, Push and Coalescing with ORIGIN

Section 10.1 of [RFC7540] uses both DNS and the presented TLS certificate to establish the origin server(s) that a connection is authoritative for, just as HTTP/1.1 does in [RFC7230].

Furthermore, Section 9.1.1 of [RFC7540] explicitly allows a connection to be used for more than one origin server, if it is authoritative. This affects what responses can be considered authoritative, both for direct responses to requests and for server push (see [RFC7540], Section 8.2.2). Indirectly, it also affects what requests will be sent on a connection, since clients will generally only send requests on connections that they believe to be authoritative for the origin in question.

Once an Origin Set has been initialised for a connection, clients that implement this specification use it to help determine what the connection is authoritative for. Specifically, such clients **MUST NOT** consider a connection to be authoritative for an origin not present in the Origin Set, and **SHOULD** use the connection for all requests to origins in the Origin Set for which the connection is authoritative, unless there are operational reasons for opening a new connection.

Note that for a connection to be considered authoritative for a given origin, the server is still required to authenticate with certificate that passes suitable checks; see Section 9.1.1 of [RFC7540] for more information. This includes verifying that the host matches a "dNSName" value from the certificate "subjectAltName" field (using the rules defined in [RFC2818]; see also [RFC5280], Section 4.2.1.6).

Additionally, clients **MAY** avoid consulting DNS to establish the connection's authority for new requests to origins in the Origin Set; however, those that do so face new risks, as explained in Section 4.

Because ORIGIN can change the set of origins a connection is used for over time, it is possible that a client might have more than one viable connection to an origin open at any time. When this occurs, clients **SHOULD NOT** emit new requests on any connection whose Origin Set is a proper subset of another connection's Origin Set, and **SHOULD** close it once all outstanding requests are satisfied.

The Origin Set is unaffected by any alternative services [RFC7838] advertisements made by the server. Advertising an alternative service does not affect whether a server is authoritative.

### 3. IANA Considerations

This specification adds an entry to the "HTTP/2 Frame Type" registry.

- o Frame Type: ORIGIN
- o Code: 0xc
- o Specification: [this document]

### 4. Security Considerations

Clients that blindly trust the ORIGIN frame's contents will be vulnerable to a large number of attacks. See Section 2.4 for mitigations.

Relaxing the requirement to consult DNS when determining authority for an origin means that an attacker who possesses a valid certificate no longer needs to be on-path to redirect traffic to them; instead of modifying DNS, they need only convince the user to visit another Web site in order to coalesce connections to the target onto their existing connection.

As a result, clients opting not to consult DNS ought to employ some alternative means to establish a high degree of confidence that the certificate is legitimate. For example, clients might skip consulting DNS only if they receive proof of inclusion in a Certificate Transparency log [RFC6962] or they have a recent OCSP response [RFC6960] (possibly using the "status\_request" TLS extension [RFC6066]) showing that the certificate was not revoked.

The Origin Set's size is unbounded by this specification, and thus could be used by attackers to exhaust client resources. To mitigate this risk, clients can monitor their state commitment and close the connection if it is too high.

### 5. References

#### 5.1. Normative References

- [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/rfc2119>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/info/rfc2818>>.

- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/info/rfc6454>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [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/rfc8174>>.

## 5.2. Informative References

- [RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <<https://www.rfc-editor.org/info/rfc6960>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.
- [RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <<https://www.rfc-editor.org/info/rfc8288>>.

### 5.3. URIs

- [1] <https://lists.w3.org/Archives/Public/ietf-http-wg/>
- [2] <http://httpwg.github.io/>
- [3] <https://github.com/httpwg/http-extensions/labels/origin-frame>

## Appendix A. Non-Normative Processing Algorithm

The following algorithm illustrates how a client could handle received ORIGIN frames:

1. If the client is configured to use a proxy for the connection, ignore the frame and stop processing.
2. If the connection is not identified with the "h2" protocol identifier or another protocol that has explicitly opted into this specification, ignore the frame and stop processing.
3. If the frame occurs upon any stream except stream 0, ignore the frame and stop processing.
4. If any of the flags 0x1, 0x2, 0x4 or 0x8 are set, ignore the frame and stop processing.
5. If no previous ORIGIN frame on the connection has reached this step, initialise the Origin Set as per Section 2.3.
6. For each "Origin-Entry" in the frame payload:
  1. Parse "ASCII-Origin" as an ASCII serialization of an origin ([RFC6454], Section 6.2) and let the result be "parsed\_origin". If parsing fails, skip to the next "Origin-Entry".
  2. Add "parsed\_origin" to the Origin Set.

## Appendix B. Operational Considerations for Servers

The ORIGIN frame allows a server to indicate for which origins a given connection ought be used. The set of origins advertised using this mechanism is under control of the server; servers are not obligated to use it, or to advertise all origins which they might be able to answer a request for.

For example, it can be used to inform the client that the connection is to only be used for the SNI-based origin, by sending an empty

ORIGIN frame. Or, a larger number of origins can be indicated by including a payload.

Generally, this information is most useful to send before sending any part of a response that might initiate a new connection; for example, "Link" header fields [RFC8288] in a response HEADERS, or links in the response body.

Therefore, the ORIGIN frame ought be sent as soon as possible on a connection, ideally before any HEADERS or PUSH\_PROMISE frames.

However, if it's desirable to associate a large number of origins with a connection, doing so might introduce end-user perceived latency, due to their size. As a result, it might be necessary to select a "core" set of origins to send initially, expanding the set of origins the connection is used for with subsequent ORIGIN frames later (e.g., when the connection is idle).

That said, senders are encouraged to include as many origins as practical within a single ORIGIN frame; clients need to make decisions about creating connections on the fly, and if the origin set is split across many frames, their behaviour might be suboptimal.

Senders take note that, as per Section 4, Step 5 of [RFC6454], the values in an ORIGIN header need to be case-normalised before serialisation.

Finally, servers that host alternative services [RFC7838] will need to explicitly advertise their origins when sending ORIGIN, because the default contents of the Origin Set (as per Section 2.3) do not contain any Alternative Services' origins, even if they have been used previously on the connection.

#### Authors' Addresses

Mark Nottingham

Email: [mnot@mnot.net](mailto:mnot@mnot.net)

URI: <https://www.mnot.net/>

Erik Nygren

Akamai

Email: [nygren@akamai.com](mailto:nygren@akamai.com)



HTTP  
Internet-Draft  
Intended status: Experimental  
Expires: September 7, 2019

C. Pratt  
D. Thakore  
CableLabs  
B. Stark  
AT&T  
March 6, 2019

HTTP Random Access and Live Content  
draft-ietf-httpbis-rand-access-live-04

Abstract

To accommodate byte range requests for content that has data appended over time, this document defines semantics that allow a HTTP client and server to perform byte-range GET and HEAD requests that start at an arbitrary byte offset within the representation and ends at an indeterminate offset.

Editorial Note (To be removed by RFC Editor before publication)

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

Working Group information can be found at <http://httpwg.github.io/>; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/rand-access-live>.

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 September 7, 2019.

## Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
1.1. Requirements Language . . . . .	3
1.2. Notational Conventions . . . . .	3
2. Performing Range requests on Random-Access Aggregating ("live") Content . . . . .	4
2.1. Establishing the Randomly Accessible Byte Range . . . . .	4
2.2. Byte-Range Requests Beyond the Randomly Accessible Byte Range . . . . .	5
3. Other Applications of Random-Access Aggregating Content . . . . .	7
3.1. Requests Starting at the Aggregation ("Live") Point . . . . .	7
3.2. Shift Buffer Representations . . . . .	8
4. Recommendations for Very Large Values . . . . .	10
5. IANA Considerations . . . . .	10
6. Security Considerations . . . . .	10
7. References . . . . .	11
7.1. Normative References . . . . .	11
7.2. Informative References . . . . .	11
Appendix A. Acknowledgements . . . . .	11
Authors' Addresses . . . . .	12

## 1. Introduction

Some Hypertext Transfer Protocol (HTTP) clients use byte-range requests (Range requests using the "bytes" Range Unit) to transfer select portions of large representations ([RFC7233]). And in some cases large representations require content to be continuously or periodically appended – such as representations consisting of live audio or video sources, blockchain databases, and log files. Clients cannot access the appended/live content using a Range request with the bytes range unit using the currently defined byte-range semantics

without accepting performance or behavior sacrifices which are not acceptable for many applications.

For instance, HTTP clients have the ability to access appended content on an indeterminate-length resource by transferring the entire representation from the beginning and continuing to read the appended content as it's made available. Obviously, this is highly inefficient for cases where the representation is large and only the most recently appended content is needed by the client.

Alternatively, clients can also access appended content by sending periodic open-ended bytes Range requests using the last-known end byte position as the range start. Performing low-frequency periodic bytes Range requests in this fashion (polling) introduces latency since the client will necessarily be somewhat behind the aggregated content - mimicking the behavior (and latency) of segmented content representations such as "HTTP Live Streaming" (HLS, [RFC8216]) or "Dynamic Adaptive Streaming over HTTP" (MPEG-DASH, [DASH]). And while performing these Range requests at higher frequency can reduce this latency, it also incurs more processing overhead and HTTP exchanges as many of the requests will return no content - since content is usually aggregated in groups of bytes (e.g. a video frame, audio sample, block, or log entry).

This document describes a usage model for range requests which enables efficient retrieval of representations that are appended to over time by using large values and associated semantics for communicating range end positions. This model allows representations to be progressively delivered by servers as new content is added. It also ensures compatibility with servers and intermediaries that don't support this technique.

### 1.1. Requirements Language

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 RFC 2119 [RFC2119].

### 1.2. Notational Conventions

This document cites productions in Augmented Backus-Naur Form (ABNF) productions from [RFC7233], using the notation defined in [RFC5234].

## 2. Performing Range requests on Random-Access Aggregating ("live") Content

This document recommends a two-step process for accessing resources that have indeterminate length representations.

Two steps are necessary because of limitations with the Range request header fields and the Content-Range response header fields. A server cannot know from a range request that a client wishes to receive a response that does not have a definite end. More critically, the header fields do not allow the server to signal that a resource has indeterminate length without also providing a fixed portion of the resource.

A client first learns that the resource has a representation of indeterminate length by requesting a range of the resource. The server responds with the range that is available, but indicates that the length of the representation is unknown using the existing Content-Range syntax. See Section 2.1 for details and examples.

Once the client knows the resource has indeterminate length, it can request a range with a very large end position from the resource. The client chooses an explicit end value larger than can be transferred in the foreseeable term. A server which supports range requests of indeterminate length signals its understanding of the client's indeterminate range request by indicating that the range it is providing has a range end that exactly matches the client's requested range end rather than a range that is bounded by what is currently available. See Section 2.2 for details.

### 2.1. Establishing the Randomly Accessible Byte Range

Establishing if a representation is continuously aggregating ("live") and determining the randomly-accessible byte range can both be determined using the existing definition for an open-ended byte-range request. Specifically, [RFC7233] defines a byte-range request of the form:

```
byte-range-spec = first-byte-pos "-" [ last-byte-pos ]
```

which allows a client to send a HEAD request with a first-byte-pos and leave last-byte-pos absent. A server that receives a satisfiable byte-range request (with first-byte-pos smaller than the current representation length) may respond with a 206 status code (Partial Content) with a Content-Range header field indicating the currently satisfiable byte range. For example:

```
HEAD /resource HTTP/1.1
Host: example.com
Range: bytes=0-
```

returns a response of the form:

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 0-1234567/*
```

from the server indicating that (1) the complete representation length is unknown (via the "\*" in place of the complete-length field) and (2) that only bytes 0-1234567 were accessible at the time the request was processed by the server. The client can infer from this response that bytes 0-1234567 of the representation can be requested and returned in a timely fashion (the bytes are immediately available).

## 2.2. Byte-Range Requests Beyond the Randomly Accessible Byte Range

Once a client has determined that a representation has an indeterminate length and established the byte range that can be accessed, it may want to perform a request with a start position within the randomly-accessible content range and an end position at an indefinite "live" point - a point where the byte-range GET request is fulfilled on-demand as the content is aggregated.

For example, for a large video asset, a client may wish to start a content transfer from the video "key" frame immediately before the point of aggregation and continue the content transfer indefinitely as content is aggregated - in order to support low-latency startup of a live video stream.

Unlike a byte-range Range request, a byte-range Content-Range response header field cannot be "open ended", per [RFC7233]:

```
byte-content-range  = bytes-unit SP
                    ( byte-range-resp / unsatisfied-range )

byte-range-resp     = byte-range "/" ( complete-length / "*" )
byte-range          = first-byte-pos "-" last-byte-pos
unsatisfied-range   = "*" / complete-length

complete-length     = 1*DIGIT
```

Specifically, last-byte-pos is required in byte-range. So in order to preserve interoperability with existing HTTP clients, servers, proxies, and caches, this document proposes a mechanism for a client

to indicate support for handling an indeterminate-length byte-range response, and a mechanism for a server to indicate if/when it's providing an indeterminate-length response.

A client can indicate support for handling indeterminate-length byte-range responses by providing a very large value for the last-byte-pos in the byte-range request. For example, a client can perform a byte-range GET request of the form:

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1230000-999999999999
```

where the last-byte-pos in the Request is much larger than the last-byte-pos returned in response to an open-ended byte-range HEAD request, as described above, and much larger than the expected maximum size of the representation. See Section 6 for range value considerations.

In response, a server may indicate that it is supplying a continuously aggregating ("live") response by supplying the client request's last-byte-pos in the Content-Range response header field.

For example:

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1230000-999999999999
```

returns

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1230000-999999999999/*
```

from the server to indicate that the response will start at byte 1230000 and continues indefinitely to include all aggregated content, as it becomes available.

A server that doesn't support or supply a continuously aggregating ("live") response will supply the currently satisfiable byte range, as it would with an open-ended byte request.

For example:

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1230000-999999999999
```

will return

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1230000-1234567/*
```

from the server to indicate that the response will start at byte 1230000 and end at byte 1234567 and will not include any aggregated content. This is the response expected from a typical HTTP server – one that doesn't support byte-range requests on aggregating content.

A client that doesn't receive a response indicating it is continuously aggregating must use other means to access aggregated content (e.g. periodic byte-range polling).

A server that does return a continuously aggregating ("live") response should return data using chunked transfer coding and not provide a Content-Length header field. A 0-length chunk indicates the end of the transfer, per [RFC7230].

### 3. Other Applications of Random-Access Aggregating Content

#### 3.1. Requests Starting at the Aggregation ("Live") Point

A client that wishes to only receive newly-aggregated portions of a resource (i.e., start at the "live" point), can use a HEAD request to learn what range the server has currently available and initiate an indeterminate-length transfer. For example:

```
HEAD /resource HTTP/1.1
Host: example.com
Range: bytes=0-
```

With the Content-Range response header field indicating the range (or ranges) available. For example:

```
206 Partial Content
Content-Range: bytes 0-1234567/*
```

The client can then issue a request for a range starting at the end value (using a very large value for the end of a range) and receive only new content.

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1234567-999999999999
```

with a server returning a Content-Range response indicating that an indeterminate-length response body will be provided

```
206 Partial Content
Content-Range: bytes 1234567-999999999999/*
```

### 3.2. Shift Buffer Representations

Some representations lend themselves to front-end content removal in addition to aggregation. While still supporting random access, representations of this type have a portion at the beginning (the "0" end) of the randomly-accessible region that become inaccessible over time. Examples of this kind of representation would be an audio-video time-shift buffer or a rolling log file.

For example a Range request containing:

```
HEAD /resource HTTP/1.1
Host: example.com
Range: bytes=0-
```

returns

```
206 Partial Content
Content-Range: bytes 1000000-1234567/*
```

indicating that the first 1000000 bytes were not accessible at the time the HEAD request was processed. Subsequent HEAD requests could return:

```
Content-Range: bytes 1000000-1234567/*
```

```
Content-Range: bytes 1010000-1244567/*
```

```
Content-Range: bytes 1020000-1254567/*
```

Note though that the difference between the first-byte-pos and last-byte-pos need not be constant.

The client could then follow-up with a GET Range request containing



```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1020000-999999999999
```

with the server returning

```
206 Partial Content
Content-Range: bytes 1020000-999999999999/*
```

with the response body returning bytes 1020000-1254567 immediately and aggregated ("live") data being returned as the content is aggregated.

A server that doesn't support or supply a continuously aggregating ("live") response will supply the currently satisfiable byte range, as it would with an open-ended byte request.

For example:

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=0-999999999999
```

will return

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1020000-1254567/*
```

from the server to indicate that the response will start at byte 1020000, end at byte 1254567, and will not include any aggregated content. This is the response expected from a typical HTTP server - one that doesn't support byte-range requests on aggregating content.

Note that responses to GET requests against shift-buffer representations using Range can be cached by intermediaries, since the Content-Range response header indicates which portion of the representation is being returned in the response body. However GET requests without a Range header cannot be cached since the first byte of the response body can vary from request to request. To ensure Range-less GET requests against shift-buffer representations are not cached, servers hosting a shift-buffer representation should either not return a 200-level response (e.g. sending a 300-level redirect response with a URI that represents the current start of the shift-buffer) or indicate the response is non-cacheable. See HTTP Caching ([RFC7234]) for details on HTTP cache control.

#### 4. Recommendations for Very Large Values

While it would be ideal to define a single numerical Very Large Value, there's no single value that would work for all applications and platforms. e.g. JavaScript numbers cannot represent all integer values above  $2^{53}$ , so a JavaScript application may want to use  $2^{53}-1$  for a Very Large Value. This value, however, would not be sufficient for all applications, such as continuously-streaming high-bitrate streams. So the value  $2^{53}-1$  (9007199254740991) is recommended as a Very Large Value unless an application has a good justification to use a smaller or larger value. e.g. If it's always known that the resource won't exceed a value smaller than the recommended Very Large Value for an application, a smaller value can be used. And if it's likely that an application will utilize resources larger than the recommended Very Large Value - such as a continuously aggregating high-bitrate media stream - a larger value should be used.

Note that, in accordance with the semantics defined above, servers that support random-access live content will need to return the last-byte-pos provided in the Range request in some cases - even if the last-byte-pos cannot be represented as a numerical value internally by the server. As is the case with any live/continuously aggregating resource, the server should terminate the content transfer when the end of the resource is reached - whether the end is due to termination of the content source or the content length exceeds the server's maximum representation length.

#### 5. IANA Considerations

This document has no actions for IANA.

#### 6. Security Considerations

As described above, servers need to be prepared to receive last-byte-pos values in Range requests that are numerically larger than the server implementation supports - and return these values in Content-Range response header fields. Servers should check the last-byte-pos value before converting and storing them into numeric form to ensure the value doesn't cause an overflow or index incorrect data. The simplest way to satisfy the live-range semantics defined in this document without potential overflow issues is to store the last-byte-pos as a string value and return it in the byte-range Content-Range response header's last-byte-pos field.

## 7. References

### 7.1. Normative References

- [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/rfc2119>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <<https://www.rfc-editor.org/info/rfc7233>>.
- [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/rfc7234>>.

### 7.2. Informative References

- [DASH] ISO, "Information technology -- Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats", ISO/IEC 23009-1:2014, May 2014, <[http://standards.iso.org/ittf/PubliclyAvailableStandards/c065274\\_ISO\\_IEC\\_23009-1\\_2014.zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c065274_ISO_IEC_23009-1_2014.zip)>.
- [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/rfc5234>>.
- [RFC8216] Pantos, R., Ed. and W. May, "HTTP Live Streaming", RFC 8216, DOI 10.17487/RFC8216, August 2017, <<https://www.rfc-editor.org/info/rfc8216>>.

## Appendix A. Acknowledgements

Mark Nottingham, Patrick McManus, Julian Reschke, Remy Lebeau, Rodger Combs, Thorsten Lohmar, Martin Thompson, Adrien de Croy, K. Morgan, Roy T. Fielding, Jeremy Poulter.

Authors' Addresses

Craig Pratt  
Portland, OR 97229  
US

Email: [pratt@acm.org](mailto:pratt@acm.org)

Darshak Thakore  
CableLabs  
858 Coal Creek Circle  
Louisville, CO 80027  
US

Email: [d.thakore@cablelabs.com](mailto:d.thakore@cablelabs.com)

Barbara Stark  
AT&T  
Atlanta, GA  
US

Email: [barbara.stark@att.com](mailto:barbara.stark@att.com)

HTTP  
Internet-Draft  
Obsoletes: 6265 (if approved)  
Intended status: Standards Track  
Expires: 26 October 2022

L. Chen, Ed.  
Google LLC  
S. Englehardt, Ed.  
Mozilla  
M. West, Ed.  
Google LLC  
J. Wilander, Ed.  
Apple, Inc  
24 April 2022

Cookies: HTTP State Management Mechanism  
draft-ietf-httpbis-rfc6265bis-10

## Abstract

This document defines the HTTP Cookie and Set-Cookie header fields. These header fields can be used by HTTP servers to store state (called cookies) at HTTP user agents, letting the servers maintain a stateful session over the mostly stateless HTTP protocol. Although cookies have many historical infelicities that degrade their security and privacy, the Cookie and Set-Cookie header fields are widely used on the Internet. This document obsoletes RFC 6265.

## About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-ietf-httpbis-rfc6265bis/>.

Discussion of this document takes place on the HTTP Working Group mailing list (<mailto:ietf-http-wg@w3.org>), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/>. Working Group information can be found at <https://httpwg.org/>.

Source for this draft and an issue tracker can be found at <https://github.com/httpwg/http-extensions/labels/6265bis>.

## 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 <https://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 26 October 2022.

## Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be 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 . . . . .	4
2. Conventions . . . . .	5
2.1. Conformance Criteria . . . . .	5
2.2. Syntax Notation . . . . .	6
2.3. Terminology . . . . .	6
3. Overview . . . . .	7
3.1. Examples . . . . .	8
4. Server Requirements . . . . .	9
4.1. Set-Cookie . . . . .	9
4.1.1. Syntax . . . . .	10
4.1.2. Semantics (Non-Normative) . . . . .	11
4.1.3. Cookie Name Prefixes . . . . .	15
4.2. Cookie . . . . .	16

4.2.1.	Syntax . . . . .	16
4.2.2.	Semantics . . . . .	16
5.	User Agent Requirements . . . . .	17
5.1.	Subcomponent Algorithms . . . . .	17
5.1.1.	Dates . . . . .	17
5.1.2.	Canonicalized Host Names . . . . .	19
5.1.3.	Domain Matching . . . . .	20
5.1.4.	Paths and Path-Match . . . . .	20
5.2.	"Same-site" and "cross-site" Requests . . . . .	21
5.2.1.	Document-based requests . . . . .	21
5.2.2.	Worker-based requests . . . . .	22
5.3.	Ignoring Set-Cookie Header Fields . . . . .	23
5.4.	The Set-Cookie Header Field . . . . .	24
5.4.1.	The Expires Attribute . . . . .	26
5.4.2.	The Max-Age Attribute . . . . .	27
5.4.3.	The Domain Attribute . . . . .	27
5.4.4.	The Path Attribute . . . . .	28
5.4.5.	The Secure Attribute . . . . .	28
5.4.6.	The HttpOnly Attribute . . . . .	28
5.4.7.	The SameSite Attribute . . . . .	28
5.5.	Storage Model . . . . .	30
5.6.	Retrieval Model . . . . .	36
5.6.1.	The Cookie Header Field . . . . .	36
5.6.2.	Non-HTTP APIs . . . . .	36
5.6.3.	Retrieval Algorithm . . . . .	37
6.	Implementation Considerations . . . . .	39
6.1.	Limits . . . . .	39
6.2.	Application Programming Interfaces . . . . .	39
6.3.	IDNA Dependency and Migration . . . . .	40
7.	Privacy Considerations . . . . .	40
7.1.	Third-Party Cookies . . . . .	41
7.2.	Cookie Policy . . . . .	41
7.3.	User Controls . . . . .	42
7.4.	Expiration Dates . . . . .	42
8.	Security Considerations . . . . .	43
8.1.	Overview . . . . .	43
8.2.	Ambient Authority . . . . .	43
8.3.	Clear Text . . . . .	44
8.4.	Session Identifiers . . . . .	44
8.5.	Weak Confidentiality . . . . .	45
8.6.	Weak Integrity . . . . .	46
8.7.	Reliance on DNS . . . . .	47
8.8.	SameSite Cookies . . . . .	47
8.8.1.	Defense in depth . . . . .	47
8.8.2.	Top-level Navigations . . . . .	47
8.8.3.	Mashups and Widgets . . . . .	48
8.8.4.	Server-controlled . . . . .	48
8.8.5.	Reload navigations . . . . .	48

8.8.6. Top-level requests with "unsafe" methods . . . . .	49
9. IANA Considerations . . . . .	50
9.1. Cookie . . . . .	50
9.2. Set-Cookie . . . . .	50
9.3. Cookie Attribute Registry . . . . .	50
9.3.1. Procedure . . . . .	51
9.3.2. Registration . . . . .	51
10. References . . . . .	51
10.1. Normative References . . . . .	51
10.2. Informative References . . . . .	53
Appendix A. Changes . . . . .	55
A.1. draft-ietf-httpbis-rfc6265bis-00 . . . . .	55
A.2. draft-ietf-httpbis-rfc6265bis-01 . . . . .	55
A.3. draft-ietf-httpbis-rfc6265bis-02 . . . . .	56
A.4. draft-ietf-httpbis-rfc6265bis-03 . . . . .	56
A.5. draft-ietf-httpbis-rfc6265bis-04 . . . . .	57
A.6. draft-ietf-httpbis-rfc6265bis-05 . . . . .	57
A.7. draft-ietf-httpbis-rfc6265bis-06 . . . . .	57
A.8. draft-ietf-httpbis-rfc6265bis-07 . . . . .	58
A.9. draft-ietf-httpbis-rfc6265bis-08 . . . . .	58
A.10. draft-ietf-httpbis-rfc6265bis-09 . . . . .	59
A.11. draft-ietf-httpbis-rfc6265bis-10 . . . . .	59
Acknowledgements . . . . .	60
Authors' Addresses . . . . .	60

## 1. Introduction

This document defines the HTTP Cookie and Set-Cookie header fields. Using the Set-Cookie header field, an HTTP server can pass name/value pairs and associated metadata (called cookies) to a user agent. When the user agent makes subsequent requests to the server, the user agent uses the metadata and other information to determine whether to return the name/value pairs in the Cookie header field.

Although simple on their surface, cookies have a number of complexities. For example, the server indicates a scope for each cookie when sending it to the user agent. The scope indicates the maximum amount of time in which the user agent should return the cookie, the servers to which the user agent should return the cookie, and the URI schemes for which the cookie is applicable.



For historical reasons, cookies contain a number of security and privacy infelicities. For example, a server can indicate that a given cookie is intended for "secure" connections, but the Secure attribute does not provide integrity in the presence of an active network attacker. Similarly, cookies for a given host are shared across all the ports on that host, even though the usual "same-origin policy" used by web browsers isolates content retrieved via different ports.

There are two audiences for this specification: developers of cookie-generating servers and developers of cookie-consuming user agents.

To maximize interoperability with user agents, servers SHOULD limit themselves to the well-behaved profile defined in Section 4 when generating cookies.

User agents MUST implement the more liberal processing rules defined in Section 5, in order to maximize interoperability with existing servers that do not conform to the well-behaved profile defined in Section 4.

This document specifies the syntax and semantics of these header fields as they are actually used on the Internet. In particular, this document does not create new syntax or semantics beyond those in use today. The recommendations for cookie generation provided in Section 4 represent a preferred subset of current server behavior, and even the more liberal cookie processing algorithm provided in Section 5 does not recommend all of the syntactic and semantic variations in use today. Where some existing software differs from the recommended protocol in significant ways, the document contains a note explaining the difference.

This document obsoletes [RFC6265].

## 2. Conventions

### 2.1. Conformance Criteria

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].

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("MUST", "SHOULD", "MAY", etc.) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant.

## 2.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

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

The OWS (optional whitespace) and BWS (bad whitespace) rules are defined in Section 5.6.3 of [HTTPSEM].

## 2.3. Terminology

The terms "user agent", "client", "server", "proxy", and "origin server" have the same meaning as in the HTTP/1.1 specification ([HTTPSEM], Section 3).

The request-host is the name of the host, as known by the user agent, to which the user agent is sending an HTTP request or from which it is receiving an HTTP response (i.e., the name of the host to which it sent the corresponding HTTP request).

The term request-uri refers to "target URI" as defined in Section 7.1 of [HTTPSEM].

Two sequences of octets are said to case-insensitively match each other if and only if they are equivalent under the `i;ascii-casemap` collation defined in [RFC4790].

The term string means a sequence of non-NUL octets.

The terms "active document", "ancestor browsing context", "browsing context", "dedicated worker", "Document", "nested browsing context", "opaque origin", "parent browsing context", "sandboxed origin browsing context flag", "shared worker", "the worker's Documents", "top-level browsing context", and "WorkerGlobalScope" are defined in [HTML].

"Service Workers" are defined in the Service Workers specification [SERVICE-WORKERS].

The term "origin", the mechanism of deriving an origin from a URI, and the "the same" matching algorithm for origins are defined in [RFC6454].

"Safe" HTTP methods include GET, HEAD, OPTIONS, and TRACE, as defined in Section 9.2.1 of [HTTPSEM].

A domain's "public suffix" is the portion of a domain that is controlled by a public registry, such as "com", "co.uk", and "pvt.k12.wy.us". A domain's "registrable domain" is the domain's public suffix plus the label to its left. That is, for `https://www.site.example`, the public suffix is `example`, and the registrable domain is `site.example`. Whenever possible, user agents SHOULD use an up-to-date public suffix list, such as the one maintained by the Mozilla project at [PSL].

The term "request", as well as a request's "client", "current url", "method", "target browsing context", and "url list", are defined in [FETCH].

The term "non-HTTP APIs" refers to non-HTTP mechanisms used to set and retrieve cookies, such as a web browser API that exposes cookies to scripts.

### 3. Overview

This section outlines a way for an origin server to send state information to a user agent and for the user agent to return the state information to the origin server.

To store state, the origin server includes a Set-Cookie header field in an HTTP response. In subsequent requests, the user agent returns a Cookie request header field to the origin server. The Cookie header field contains cookies the user agent received in previous Set-Cookie header fields. The origin server is free to ignore the Cookie header field or use its contents for an application-defined purpose.

Origin servers MAY send a Set-Cookie response header field with any response. An origin server can include multiple Set-Cookie header fields in a single response. The presence of a Cookie or a Set-Cookie header field does not preclude HTTP caches from storing and reusing a response.

Origin servers SHOULD NOT fold multiple Set-Cookie header fields into a single header field. The usual mechanism for folding HTTP headers fields (i.e., as defined in Section 5.3 of [HTTPSEM]) might change the semantics of the Set-Cookie header field because the %x2C (",") character is used by Set-Cookie in a way that conflicts with such folding.

User agents MAY ignore Set-Cookie header fields based on response status codes or the user agent's cookie policy (see Section 5.3).

### 3.1. Examples

Using the Set-Cookie header field, a server can send the user agent a short string in an HTTP response that the user agent will return in future HTTP requests that are within the scope of the cookie. For example, the server can send the user agent a "session identifier" named SID with the value 31d4d96e407aad42. The user agent then returns the session identifier in subsequent requests.

```
== Server -> User Agent ==
```

```
Set-Cookie: SID=31d4d96e407aad42
```

```
== User Agent -> Server ==
```

```
Cookie: SID=31d4d96e407aad42
```

The server can alter the default scope of the cookie using the Path and Domain attributes. For example, the server can instruct the user agent to return the cookie to every path and every subdomain of site.example.

```
== Server -> User Agent ==
```

```
Set-Cookie: SID=31d4d96e407aad42; Path=/; Domain=site.example
```

```
== User Agent -> Server ==
```

```
Cookie: SID=31d4d96e407aad42
```

As shown in the next example, the server can store multiple cookies at the user agent. For example, the server can store a session identifier as well as the user's preferred language by returning two Set-Cookie header fields. Notice that the server uses the Secure and HttpOnly attributes to provide additional security protections for the more sensitive session identifier (see Section 4.1.2).

== Server -> User Agent ==

```
Set-Cookie: SID=31d4d96e407aad42; Path=/; Secure; HttpOnly
Set-Cookie: lang=en-US; Path=/; Domain=site.example
```

== User Agent -> Server ==

```
Cookie: SID=31d4d96e407aad42; lang=en-US
```

Notice that the Cookie header field above contains two cookies, one named SID and one named lang. If the server wishes the user agent to persist the cookie over multiple "sessions" (e.g., user agent restarts), the server can specify an expiration date in the Expires attribute. Note that the user agent might delete the cookie before the expiration date if the user agent's cookie store exceeds its quota or if the user manually deletes the server's cookie.

== Server -> User Agent ==

```
Set-Cookie: lang=en-US; Expires=Wed, 09 Jun 2021 10:18:14 GMT
```

== User Agent -> Server ==

```
Cookie: SID=31d4d96e407aad42; lang=en-US
```

Finally, to remove a cookie, the server returns a Set-Cookie header field with an expiration date in the past. The server will be successful in removing the cookie only if the Path and the Domain attribute in the Set-Cookie header field match the values used when the cookie was created.

== Server -> User Agent ==

```
Set-Cookie: lang=; Expires=Sun, 06 Nov 1994 08:49:37 GMT
```

== User Agent -> Server ==

```
Cookie: SID=31d4d96e407aad42
```

#### 4. Server Requirements

This section describes the syntax and semantics of a well-behaved profile of the Cookie and Set-Cookie header fields.

##### 4.1. Set-Cookie

The Set-Cookie HTTP response header field is used to send cookies from the server to the user agent.

## 4.1.1. Syntax

Informally, the Set-Cookie response header field contains a cookie, which begins with a name-value-pair, followed by zero or more attribute-value pairs. Servers SHOULD NOT send Set-Cookie header fields that fail to conform to the following grammar:

```

set-cookie           = set-cookie-string
set-cookie-string    = BWS cookie-pair *( BWS ";" OWS cookie-av )
cookie-pair          = cookie-name BWS "=" BWS cookie-value
cookie-name          = 1*cookie-octet
cookie-value         = *cookie-octet / ( DQUOTE *cookie-octet DQUOTE )
cookie-octet         = %x21 / %x23-2B / %x2D-3A / %x3C-5B / %x5D-7E
                      / %x80-FF
                      ; octets excluding CTLs,
                      ; whitespace DQUOTE, comma, semicolon,
                      ; and backslash

cookie-av            = expires-av / max-age-av / domain-av /
                      path-av / secure-av / httponly-av /
                      samesite-av / extension-av
expires-av           = "Expires" BWS "=" BWS sane-cookie-date
sane-cookie-date     =
    <IMF-fixdate, defined in [HTTPSEM], Section 5.6.7>
max-age-av           = "Max-Age" BWS "=" BWS non-zero-digit *DIGIT
non-zero-digit       = %x31-39
                      ; digits 1 through 9
domain-av            = "Domain" BWS "=" BWS domain-value
domain-value         = <subdomain>
                      ; see details below
path-av              = "Path" BWS "=" BWS path-value
path-value           = *av-octet
secure-av            = "Secure"
httponly-av          = "HttpOnly"
samesite-av          = "SameSite" BWS "=" BWS samesite-value
samesite-value       = "Strict" / "Lax" / "None"
extension-av         = *av-octet
av-octet             = %x20-3A / %x3C-7E
                      ; any CHAR except CTLs or ";"

```

Note that some of the grammatical terms above reference documents that use different grammatical notations than this document (which uses ABNF from [RFC5234]).

The semantics of the cookie-value are not defined by this document.

To maximize compatibility with user agents, servers that wish to store arbitrary data in a cookie-value SHOULD encode that data, for example, using Base64 [RFC4648].

The domain-value is a subdomain as defined by [RFC1034], Section 3.5, and as enhanced by [RFC1123], Section 2.1. Thus, domain-value is a string of [USASCII] characters, such as one obtained by applying the "ToASCII" operation defined in Section 4 of [RFC3490].

Per the grammar above, the cookie-value MAY be wrapped in DQUOTE characters. Note that in this case, the initial and trailing DQUOTE characters are not stripped. They are part of the cookie-value, and will be included in Cookie header fields sent to the server.

The portions of the set-cookie-string produced by the cookie-av term are known as attributes. To maximize compatibility with user agents, servers SHOULD NOT produce two attributes with the same name in the same set-cookie-string. (See Section 5.5 for how user agents handle this case.)

Servers SHOULD NOT include more than one Set-Cookie header field in the same response with the same cookie-name. (See Section 5.4 for how user agents handle this case.)

If a server sends multiple responses containing Set-Cookie header fields concurrently to the user agent (e.g., when communicating with the user agent over multiple sockets), these responses create a "race condition" that can lead to unpredictable behavior.

NOTE: Some existing user agents differ in their interpretation of two-digit years. To avoid compatibility issues, servers SHOULD use the rfc1123-date format, which requires a four-digit year.

NOTE: Some user agents store and process dates in cookies as 32-bit UNIX time\_t values. Implementation bugs in the libraries supporting time\_t processing on some systems might cause such user agents to process dates after the year 2038 incorrectly.

#### 4.1.2. Semantics (Non-Normative)

This section describes simplified semantics of the Set-Cookie header field. These semantics are detailed enough to be useful for understanding the most common uses of cookies by servers. The full semantics are described in Section 5.

When the user agent receives a Set-Cookie header field, the user agent stores the cookie together with its attributes. Subsequently, when the user agent makes an HTTP request, the user agent includes the applicable, non-expired cookies in the Cookie header field.

If the user agent receives a new cookie with the same cookie-name, domain-value, and path-value as a cookie that it has already stored, the existing cookie is evicted and replaced with the new cookie. Notice that servers can delete cookies by sending the user agent a new cookie with an Expires attribute with a value in the past.

Unless the cookie's attributes indicate otherwise, the cookie is returned only to the origin server (and not, for example, to any subdomains), and it expires at the end of the current session (as defined by the user agent). User agents ignore unrecognized cookie attributes (but not the entire cookie).

#### 4.1.2.1. The Expires Attribute

The Expires attribute indicates the maximum lifetime of the cookie, represented as the date and time at which the cookie expires. The user agent is not required to retain the cookie until the specified date has passed. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

The user agent MUST limit the maximum value of the Expires attribute. The limit SHOULD NOT be greater than 400 days (34560000 seconds) in the future. The RECOMMENDED limit is 400 days in the future, but the user agent MAY adjust the limit (see Section 7.2). Expires attributes that are greater than the limit MUST be reduced to the limit.

#### 4.1.2.2. The Max-Age Attribute

The Max-Age attribute indicates the maximum lifetime of the cookie, represented as the number of seconds until the cookie expires. The user agent is not required to retain the cookie for the specified duration. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

The user agent MUST limit the maximum value of the Max-Age attribute. The limit SHOULD NOT be greater than 400 days (34560000 seconds) in duration. The RECOMMENDED limit is 400 days in duration, but the user agent MAY adjust the limit (see Section 7.2). Max-Age attributes that are greater than the limit MUST be reduced to the limit.



NOTE: Some existing user agents do not support the Max-Age attribute. User agents that do not support the Max-Age attribute ignore the attribute.

If a cookie has both the Max-Age and the Expires attribute, the Max-Age attribute has precedence and controls the expiration date of the cookie. If a cookie has neither the Max-Age nor the Expires attribute, the user agent will retain the cookie until "the current session is over" (as defined by the user agent).

#### 4.1.2.3. The Domain Attribute

The Domain attribute specifies those hosts to which the cookie will be sent. For example, if the value of the Domain attribute is "site.example", the user agent will include the cookie in the Cookie header field when making HTTP requests to site.example, www.site.example, and www.corp.site.example. (Note that a leading %x2E ("."), if present, is ignored even though that character is not permitted, but a trailing %x2E ("."), if present, will cause the user agent to ignore the attribute.) If the server omits the Domain attribute, the user agent will return the cookie only to the origin server.

WARNING: Some existing user agents treat an absent Domain attribute as if the Domain attribute were present and contained the current host name. For example, if site.example returns a Set-Cookie header field without a Domain attribute, these user agents will erroneously send the cookie to www.site.example as well.

The user agent will reject cookies unless the Domain attribute specifies a scope for the cookie that would include the origin server. For example, the user agent will accept a cookie with a Domain attribute of "site.example" or of "foo.site.example" from foo.site.example, but the user agent will not accept a cookie with a Domain attribute of "bar.site.example" or of "baz.foo.site.example".

NOTE: For security reasons, many user agents are configured to reject Domain attributes that correspond to "public suffixes". For example, some user agents will reject Domain attributes of "com" or "co.uk". (See Section 5.5 for more information.)

#### 4.1.2.4. The Path Attribute

The scope of each cookie is limited to a set of paths, controlled by the Path attribute. If the server omits the Path attribute, the user agent will use the "directory" of the request-uri's path component as the default value. (See Section 5.1.4 for more details.)

The user agent will include the cookie in an HTTP request only if the path portion of the request-uri matches (or is a subdirectory of) the cookie's Path attribute, where the %x2F ("/") character is interpreted as a directory separator.

Although seemingly useful for isolating cookies between different paths within a given host, the Path attribute cannot be relied upon for security (see Section 8).

#### 4.1.2.5. The Secure Attribute

The Secure attribute limits the scope of the cookie to "secure" channels (where "secure" is defined by the user agent). When a cookie has the Secure attribute, the user agent will include the cookie in an HTTP request only if the request is transmitted over a secure channel (typically HTTP over Transport Layer Security (TLS) [RFC2818]).

#### 4.1.2.6. The HttpOnly Attribute

The HttpOnly attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via non-HTTP APIs.

Note that the HttpOnly attribute is independent of the Secure attribute: a cookie can have both the HttpOnly and the Secure attribute.

#### 4.1.2.7. The SameSite Attribute

The "SameSite" attribute limits the scope of the cookie such that it will only be attached to requests if those requests are same-site, as defined by the algorithm in Section 5.2. For example, requests for `https://site.example/sekrit-image` will attach same-site cookies if and only if initiated from a context whose "site for cookies" is an origin with a scheme and registered domain of "https" and "site.example" respectively.

If the "SameSite" attribute's value is "Strict", the cookie will only be sent along with "same-site" requests. If the value is "Lax", the cookie will be sent with same-site requests, and with "cross-site" top-level navigations, as described in Section 5.4.7.1. If the value is "None", the cookie will be sent with same-site and cross-site requests. If the "SameSite" attribute's value is something other than these three known keywords, the attribute's value will be subject to a default enforcement mode that is equivalent to "Lax".

The "SameSite" attribute affects cookie creation as well as delivery. Cookies which assert "SameSite=Lax" or "SameSite=Strict" cannot be set in responses to cross-site subresource requests, or cross-site nested navigations. They can be set along with any top-level navigation, cross-site or otherwise.

#### 4.1.3. Cookie Name Prefixes

Section 8.5 and Section 8.6 of this document spell out some of the drawbacks of cookies' historical implementation. In particular, it is impossible for a server to have confidence that a given cookie was set with a particular set of attributes. In order to provide such confidence in a backwards-compatible way, two common sets of requirements can be inferred from the first few characters of the cookie's name.

The normative requirements for the prefixes described below are detailed in the storage model algorithm defined in Section 5.5.

##### 4.1.3.1. The "\_\_Secure-" Prefix

If a cookie's name begins with a case-sensitive match for the string `__Secure-`, then the cookie will have been set with a Secure attribute.

For example, the following Set-Cookie header field would be rejected by a conformant user agent, as it does not have a Secure attribute.

```
Set-Cookie: __Secure-SID=12345; Domain=site.example
```

Whereas the following Set-Cookie header field would be accepted if set from a secure origin (e.g. "https://site.example/"), and rejected otherwise:

```
Set-Cookie: __Secure-SID=12345; Domain=site.example; Secure
```

##### 4.1.3.2. The "\_\_Host-" Prefix

If a cookie's name begins with a case-sensitive match for the string `__Host-`, then the cookie will have been set with a Secure attribute, a Path attribute with a value of `/`, and no Domain attribute.

This combination yields a cookie that hews as closely as a cookie can to treating the origin as a security boundary. The lack of a Domain attribute ensures that the cookie's host-only-flag is true, locking the cookie to a particular host, rather than allowing it to span subdomains. Setting the Path to / means that the cookie is effective for the entire host, and won't be overridden for specific paths. The Secure attribute ensures that the cookie is unaltered by non-secure origins, and won't span protocols.

Ports are the only piece of the origin model that `__Host-` cookies continue to ignore.

For example, the following cookies would always be rejected:

```
Set-Cookie: __Host-SID=12345
Set-Cookie: __Host-SID=12345; Secure
Set-Cookie: __Host-SID=12345; Domain=site.example
Set-Cookie: __Host-SID=12345; Domain=site.example; Path=/
Set-Cookie: __Host-SID=12345; Secure; Domain=site.example; Path=/
```

While the following would be accepted if set from a secure origin (e.g. "https://site.example/"), and rejected otherwise:

```
Set-Cookie: __Host-SID=12345; Secure; Path=/
```

## 4.2. Cookie

### 4.2.1. Syntax

The user agent sends stored cookies to the origin server in the Cookie header field. If the server conforms to the requirements in Section 4.1 (and the user agent conforms to the requirements in Section 5), the user agent will send a Cookie header field that conforms to the following grammar:

```
cookie           = cookie-string
cookie-string    = cookie-pair *( ";" SP cookie-pair )
```

### 4.2.2. Semantics

Each cookie-pair represents a cookie stored by the user agent. The cookie-pair contains the cookie-name and cookie-value the user agent received in the Set-Cookie header field.

Notice that the cookie attributes are not returned. In particular, the server cannot determine from the Cookie field alone when a cookie will expire, for which hosts the cookie is valid, for which paths the cookie is valid, or whether the cookie was set with the Secure or HttpOnly attributes.

The semantics of individual cookies in the Cookie header field are not defined by this document. Servers are expected to imbue these cookies with application-specific semantics.

Although cookies are serialized linearly in the Cookie header field, servers SHOULD NOT rely upon the serialization order. In particular, if the Cookie header field contains two cookies with the same name (e.g., that were set with different Path or Domain attributes), servers SHOULD NOT rely upon the order in which these cookies appear in the header field.

## 5. User Agent Requirements

This section specifies the Cookie and Set-Cookie header fields in sufficient detail that a user agent implementing these requirements precisely can interoperate with existing servers (even those that do not conform to the well-behaved profile described in Section 4).

A user agent could enforce more restrictions than those specified herein (e.g., restrictions specified by its cookie policy, described in Section 7.2). However, such additional restrictions may reduce the likelihood that a user agent will be able to interoperate with existing servers.

### 5.1. Subcomponent Algorithms

This section defines some algorithms used by user agents to process specific subcomponents of the Cookie and Set-Cookie header fields.

#### 5.1.1. Dates

The user agent MUST use an algorithm equivalent to the following algorithm to parse a cookie-date. Note that the various boolean flags defined as a part of the algorithm (i.e., found-time, found-day-of-month, found-month, found-year) are initially "not set".

1. Using the grammar below, divide the cookie-date into date-tokens.

```

cookie-date      = *delimiter date-token-list *delimiter
date-token-list = date-token *( 1*delimiter date-token )
date-token      = 1*non-delimiter

delimiter        = %x09 / %x20-2F / %x3B-40 / %x5B-60 / %x7B-7E
non-delimiter    = %x00-08 / %x0A-1F / DIGIT / ":" / ALPHA
                  / %x7F-FF
non-digit        = %x00-2F / %x3A-FF

day-of-month     = 1*2DIGIT [ non-digit *OCTET ]
month            = ( "jan" / "feb" / "mar" / "apr" /
                    "may" / "jun" / "jul" / "aug" /
                    "sep" / "oct" / "nov" / "dec" ) *OCTET
year             = 2*4DIGIT [ non-digit *OCTET ]
time             = hms-time [ non-digit *OCTET ]
hms-time         = time-field ":" time-field ":" time-field
time-field       = 1*2DIGIT

```

2. Process each date-token sequentially in the order the date-tokens appear in the cookie-date:
  1. If the found-time flag is not set and the token matches the time production, set the found-time flag and set the hour-value, minute-value, and second-value to the numbers denoted by the digits in the date-token, respectively. Skip the remaining sub-steps and continue to the next date-token.
  2. If the found-day-of-month flag is not set and the date-token matches the day-of-month production, set the found-day-of-month flag and set the day-of-month-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.
  3. If the found-month flag is not set and the date-token matches the month production, set the found-month flag and set the month-value to the month denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.
  4. If the found-year flag is not set and the date-token matches the year production, set the found-year flag and set the year-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.
3. If the year-value is greater than or equal to 70 and less than or equal to 99, increment the year-value by 1900.
4. If the year-value is greater than or equal to 0 and less than or equal to 69, increment the year-value by 2000.

1. NOTE: Some existing user agents interpret two-digit years differently.
5. Abort these steps and fail to parse the cookie-date if:
  - \* at least one of the found-day-of-month, found-month, found-year, or found-time flags is not set,
  - \* the day-of-month-value is less than 1 or greater than 31,
  - \* the year-value is less than 1601,
  - \* the hour-value is greater than 23,
  - \* the minute-value is greater than 59, or
  - \* the second-value is greater than 59.

(Note that leap seconds cannot be represented in this syntax.)
6. Let the parsed-cookie-date be the date whose day-of-month, month, year, hour, minute, and second (in UTC) are the day-of-month-value, the month-value, the year-value, the hour-value, the minute-value, and the second-value, respectively. If no such date exists, abort these steps and fail to parse the cookie-date.
7. Return the parsed-cookie-date as the result of this algorithm.

#### 5.1.2. Canonicalized Host Names

A canonicalized host name is the string generated by the following algorithm:

1. Convert the host name to a sequence of individual domain name labels.
2. Convert each label that is not a Non-Reserved LDH (NR-LDH) label, to an A-label (see Section 2.3.2.1 of [RFC5890] for the former and latter), or to a "punycode label" (a label resulting from the "ToASCII" conversion in Section 4 of [RFC3490]), as appropriate (see Section 6.3 of this specification).
3. Concatenate the resulting labels, separated by a %x2E (".") character.

### 5.1.3. Domain Matching

A string domain-matches a given domain string if at least one of the following conditions hold:

- \* The domain string and the string are identical. (Note that both the domain string and the string will have been canonicalized to lower case at this point.)
- \* All of the following conditions hold:
  - The domain string is a suffix of the string.
  - The last character of the string that is not included in the domain string is a %x2E (".") character.
  - The string is a host name (i.e., not an IP address).

### 5.1.4. Paths and Path-Match

The user agent MUST use an algorithm equivalent to the following algorithm to compute the default-path of a cookie:

1. Let uri-path be the path portion of the request-uri if such a portion exists (and empty otherwise).
2. If the uri-path is empty or if the first character of the uri-path is not a %x2F ("/") character, output %x2F ("/") and skip the remaining steps.
3. If the uri-path contains no more than one %x2F ("/") character, output %x2F ("/") and skip the remaining step.
4. Output the characters of the uri-path from the first character up to, but not including, the right-most %x2F ("/").

A request-path path-matches a given cookie-path if at least one of the following conditions holds:

- \* The cookie-path and the request-path are identical.

Note that this differs from the rules in [RFC3986] for equivalence of the path component, and hence two equivalent paths can have different cookies.

- \* The cookie-path is a prefix of the request-path, and the last character of the cookie-path is %x2F ("/").



- \* The cookie-path is a prefix of the request-path, and the first character of the request-path that is not included in the cookie-path is a %x2F ("/") character.

## 5.2. "Same-site" and "cross-site" Requests

Two origins are same-site if they satisfy the "same site" criteria defined in [SAMESITE]. A request is "same-site" if the following criteria are true:

1. The request is not the result of a cross-site redirect. That is, the origin of every url in the request's url list is same-site with the request's current url's origin.
2. The request is not the result of a reload navigation triggered through a user interface element (as defined by the user agent; e.g., a request triggered by the user clicking a refresh button on a toolbar).
3. The request's current url's origin is same-site with the request's client's "site for cookies" (which is an origin), or if the request has no client or the request's client is null.

Requests which are the result of a reload navigation triggered through a user interface element are same-site if the reloaded document was originally navigated to via a same-site request. A request that is not "same-site" is instead "cross-site".

The request's client's "site for cookies" is calculated depending upon its client's type, as described in the following subsections:

### 5.2.1. Document-based requests

The URI displayed in a user agent's address bar is the only security context directly exposed to users, and therefore the only signal users can reasonably rely upon to determine whether or not they trust a particular website. The origin of that URI represents the context in which a user most likely believes themselves to be interacting. We'll define this origin, the top-level browsing context's active document's origin, as the "top-level origin".

For a document displayed in a top-level browsing context, we can stop here: the document's "site for cookies" is the top-level origin.

For documents which are displayed in nested browsing contexts, we need to audit the origins of each of a document's ancestor browsing contexts' active documents in order to account for the "multiple-nested scenarios" described in Section 4 of [RFC7034]. A document's

"site for cookies" is the top-level origin if and only if the top-level origin is same-site with the document's origin, and with each of the document's ancestor documents' origins. Otherwise its "site for cookies" is an origin set to an opaque origin.

Given a Document (document), the following algorithm returns its "site for cookies":

1. Let top-document be the active document in document's browsing context's top-level browsing context.
2. Let top-origin be the origin of top-document's URI if top-document's sandboxed origin browsing context flag is set, and top-document's origin otherwise.
3. Let documents be a list containing document and each of document's ancestor browsing contexts' active documents.
4. For each item in documents:
  1. Let origin be the origin of item's URI if item's sandboxed origin browsing context flag is set, and item's origin otherwise.
  2. If origin is not same-site with top-origin, return an origin set to an opaque origin.
5. Return top-origin.

#### 5.2.2. Worker-based requests

Worker-driven requests aren't as clear-cut as document-driven requests, as there isn't a clear link between a top-level browsing context and a worker. This is especially true for Service Workers [SERVICE-WORKERS], which may execute code in the background, without any document visible at all.

Note: The descriptions below assume that workers must be same-origin with the documents that instantiate them. If this invariant changes, we'll need to take the worker's script's URI into account when determining their status.

##### 5.2.2.1. Dedicated and Shared Workers

Dedicated workers are simple, as each dedicated worker is bound to one and only one document. Requests generated from a dedicated worker (via `importScripts`, `XMLHttpRequest`, `fetch()`, etc) define their "site for cookies" as that document's "site for cookies".

Shared workers may be bound to multiple documents at once. As it is quite possible for those documents to have distinct "site for cookies" values, the worker's "site for cookies" will be an origin set to an opaque origin in cases where the values are not all same-site with the worker's origin, and the worker's origin in cases where the values agree.

Given a `WorkerGlobalScope` (`worker`), the following algorithm returns its "site for cookies":

1. Let `site` be `worker`'s origin.
2. For each document in `worker`'s Documents:
  1. Let `document-site` be document's "site for cookies" (as defined in Section 5.2.1).
  2. If `document-site` is not same-site with `site`, return an origin set to an opaque origin.
3. Return `site`.

#### 5.2.2.2. Service Workers

Service Workers are more complicated, as they act as a completely separate execution context with only tangential relationship to the Document which registered them.

Requests which simply pass through a Service Worker will be handled as described above: the request's client will be the Document or Worker which initiated the request, and its "site for cookies" will be those defined in Section 5.2.1 and Section 5.2.2.1

Requests which are initiated by the Service Worker itself (via a direct call to `fetch()`, for instance), on the other hand, will have a client which is a `ServiceWorkerGlobalScope`. Its "site for cookies" will be the Service Worker's URI's origin.

Given a `ServiceWorkerGlobalScope` (`worker`), the following algorithm returns its "site for cookies":

1. Return `worker`'s origin.

#### 5.3. Ignoring Set-Cookie Header Fields

User agents MAY ignore Set-Cookie header fields contained in responses with 100-level status codes or based on its cookie policy (see Section 7.2).

All other Set-Cookie header fields SHOULD be processed according to Section 5.4. That is, Set-Cookie header fields contained in responses with non-100-level status codes (including those in responses with 400- and 500-level status codes) SHOULD be processed unless ignored according to the user agent's cookie policy.

#### 5.4. The Set-Cookie Header Field

When a user agent receives a Set-Cookie header field in an HTTP response, the user agent MAY ignore the Set-Cookie header field in its entirety (see Section 5.3).

If the user agent does not ignore the Set-Cookie header field in its entirety, the user agent MUST parse the field-value of the Set-Cookie header field as a set-cookie-string (defined below).

NOTE: The algorithm below is more permissive than the grammar in Section 4.1. For example, the algorithm strips leading and trailing whitespace from the cookie name and value (but maintains internal whitespace), whereas the grammar in Section 4.1 forbids whitespace in these positions. In addition, the algorithm below accommodates some characters that are not cookie-octets according to the grammar in Section 4.1. User agents use this algorithm so as to interoperate with servers that do not follow the recommendations in Section 4.

NOTE: As set-cookie-string may originate from a non-HTTP API, it is not guaranteed to be free of CTL characters, so this algorithm handles them explicitly. Horizontal tab (%x09) is excluded from the CTL characters that lead to set-cookie-string rejection, as it is considered whitespace, which is handled separately.

NOTE: The set-cookie-string may contain octet sequences that appear percent-encoded as per Section 2.1 of [RFC3986]. However, a user agent MUST NOT decode these sequences and instead parse the individual octets as specified in this algorithm.

A user agent MUST use an algorithm equivalent to the following algorithm to parse a set-cookie-string:

1. If the set-cookie-string contains a %x00-08 / %x0A-1F / %x7F character (CTL characters excluding HTAB): Abort these steps and ignore the set-cookie-string entirely.
2. If the set-cookie-string contains a %x3B (";") character:

1. The name-value-pair string consists of the characters up to, but not including, the first %x3B (";"), and the unparsed-attributes consist of the remainder of the set-cookie-string (including the %x3B (";") in question).

Otherwise:

1. The name-value-pair string consists of all the characters contained in the set-cookie-string, and the unparsed-attributes is the empty string.
3. If the name-value-pair string lacks a %x3D ("=") character, then the name string is empty, and the value string is the value of name-value-pair.

Otherwise, the name string consists of the characters up to, but not including, the first %x3D ("=") character, and the (possibly empty) value string consists of the characters after the first %x3D ("=") character.

4. Remove any leading or trailing WSP characters from the name string and the value string.
5. If the sum of the lengths of the name string and the value string is more than 4096 octets, abort these steps and ignore the set-cookie-string entirely.
6. The cookie-name is the name string, and the cookie-value is the value string.

The user agent MUST use an algorithm equivalent to the following algorithm to parse the unparsed-attributes:

1. If the unparsed-attributes string is empty, skip the rest of these steps.
2. Discard the first character of the unparsed-attributes (which will be a %x3B (";") character).
3. If the remaining unparsed-attributes contains a %x3B (";") character:
  1. Consume the characters of the unparsed-attributes up to, but not including, the first %x3B (";") character.

Otherwise:

1. Consume the remainder of the unparsed-attributes.

Let the cookie-av string be the characters consumed in this step.

4. If the cookie-av string contains a %x3D ("=") character:
  1. The (possibly empty) attribute-name string consists of the characters up to, but not including, the first %x3D ("=") character, and the (possibly empty) attribute-value string consists of the characters after the first %x3D ("=") character.
- Otherwise:
  1. The attribute-name string consists of the entire cookie-av string, and the attribute-value string is empty.
5. Remove any leading or trailing WSP characters from the attribute-name string and the attribute-value string.
6. If the attribute-value is longer than 1024 octets, ignore the cookie-av string and return to Step 1 of this algorithm.
7. Process the attribute-name and attribute-value according to the requirements in the following subsections. (Notice that attributes with unrecognized attribute-names are ignored.)
8. Return to Step 1 of this algorithm.

When the user agent finishes parsing the set-cookie-string, the user agent is said to "receive a cookie" from the request-uri with name cookie-name, value cookie-value, and attributes cookie-attribute-list. (See Section 5.5 for additional requirements triggered by receiving a cookie.)

#### 5.4.1. The Expires Attribute

If the attribute-name case-insensitively matches the string "Expires", the user agent MUST process the cookie-av as follows.

1. Let the expiry-time be the result of parsing the attribute-value as cookie-date (see Section 5.1.1).
2. If the attribute-value failed to parse as a cookie date, ignore the cookie-av.
3. Let cookie-age-limit be the maximum age of the cookie (which SHOULD be 400 days in the future or sooner, see Section 4.1.2.1).

4. If the expiry-time is more than cookie-age-limit, the user agent MUST set the expiry time to cookie-age-limit in seconds.
5. If the expiry-time is earlier than the earliest date the user agent can represent, the user agent MAY replace the expiry-time with the earliest representable date.
6. Append an attribute to the cookie-attribute-list with an attribute-name of Expires and an attribute-value of expiry-time.

#### 5.4.2. The Max-Age Attribute

If the attribute-name case-insensitively matches the string "Max-Age", the user agent MUST process the cookie-av as follows.

1. If the first character of the attribute-value is not a DIGIT or a "-" character, ignore the cookie-av.
2. If the remainder of attribute-value contains a non-DIGIT character, ignore the cookie-av.
3. Let delta-seconds be the attribute-value converted to an integer.
4. Let cookie-age-limit be the maximum age of the cookie (which SHOULD be 400 days or less, see Section 4.1.2.2).
5. Set delta-seconds to the smaller of its present value and cookie-age-limit.
6. If delta-seconds is less than or equal to zero (0), let expiry-time be the earliest representable date and time. Otherwise, let the expiry-time be the current date and time plus delta-seconds seconds.
7. Append an attribute to the cookie-attribute-list with an attribute-name of Max-Age and an attribute-value of expiry-time.

#### 5.4.3. The Domain Attribute

If the attribute-name case-insensitively matches the string "Domain", the user agent MUST process the cookie-av as follows.

1. Let cookie-domain be the attribute-value.
2. If cookie-domain starts with %x2E ("."), let cookie-domain be cookie-domain without its leading %x2E (".").
3. Convert the cookie-domain to lower case.

4. Append an attribute to the cookie-attribute-list with an attribute-name of Domain and an attribute-value of cookie-domain.

#### 5.4.4. The Path Attribute

If the attribute-name case-insensitively matches the string "Path", the user agent MUST process the cookie-av as follows.

1. If the attribute-value is empty or if the first character of the attribute-value is not %x2F ("/"):

1. Let cookie-path be the default-path.

Otherwise:

1. Let cookie-path be the attribute-value.

2. Append an attribute to the cookie-attribute-list with an attribute-name of Path and an attribute-value of cookie-path.

#### 5.4.5. The Secure Attribute

If the attribute-name case-insensitively matches the string "Secure", the user agent MUST append an attribute to the cookie-attribute-list with an attribute-name of Secure and an empty attribute-value.

#### 5.4.6. The HttpOnly Attribute

If the attribute-name case-insensitively matches the string "HttpOnly", the user agent MUST append an attribute to the cookie-attribute-list with an attribute-name of HttpOnly and an empty attribute-value.

#### 5.4.7. The SameSite Attribute

If the attribute-name case-insensitively matches the string "SameSite", the user agent MUST process the cookie-av as follows:

1. Let enforcement be "Default".
2. If cookie-av's attribute-value is a case-insensitive match for "None", set enforcement to "None".
3. If cookie-av's attribute-value is a case-insensitive match for "Strict", set enforcement to "Strict".
4. If cookie-av's attribute-value is a case-insensitive match for "Lax", set enforcement to "Lax".



5. Append an attribute to the cookie-attribute-list with an attribute-name of "SameSite" and an attribute-value of enforcement.

#### 5.4.7.1. "Strict" and "Lax" enforcement

Same-site cookies in "Strict" enforcement mode will not be sent along with top-level navigations which are triggered from a cross-site document context. As discussed in Section 8.8.2, this might or might not be compatible with existing session management systems. In the interests of providing a drop-in mechanism that mitigates the risk of CSRF attacks, developers may set the SameSite attribute in a "Lax" enforcement mode that carves out an exception which sends same-site cookies along with cross-site requests if and only if they are top-level navigations which use a "safe" (in the [HTTPSEM] sense) HTTP method. (Note that a request's method may be changed from POST to GET for some redirects (see Sections 15.4.2 and 15.4.3 of [HTTPSEM]); in these cases, a request's "safe"ness is determined based on the method of the current redirect hop.)

Lax enforcement provides reasonable defense in depth against CSRF attacks that rely on unsafe HTTP methods (like POST), but does not offer a robust defense against CSRF as a general category of attack:

1. Attackers can still pop up new windows or trigger top-level navigations in order to create a "same-site" request (as described in Section 5.2.1), which is only a speedbump along the road to exploitation.
2. Features like <link rel='prerender'> [prerendering] can be exploited to create "same-site" requests without the risk of user detection.

When possible, developers should use a session management mechanism such as that described in Section 8.8.2 to mitigate the risk of CSRF more completely.

#### 5.4.7.2. "Lax-Allowing-Unsafe" enforcement

As discussed in Section 8.8.6, compatibility concerns may necessitate the use of a "Lax-allowing-unsafe" enforcement mode that allows cookies to be sent with a cross-site HTTP request if and only if it is a top-level request, regardless of request method. That is, the "Lax-allowing-unsafe" enforcement mode waives the requirement for the HTTP request's method to be "safe" in the SameSite enforcement step of the retrieval algorithm in Section 5.6.3. (All cookies, regardless of SameSite enforcement mode, may be set for top-level navigations, regardless of HTTP request method, as specified in

Section 5.5.)

"Lax-allowing-unsafe" is not a distinct value of the SameSite attribute. Rather, user agents MAY apply "Lax-allowing-unsafe" enforcement only to cookies that did not explicitly specify a SameSite attribute (i.e., those whose same-site-flag was set to "Default" by default). To limit the scope of this compatibility mode, user agents which apply "Lax-allowing-unsafe" enforcement SHOULD restrict the enforcement to cookies which were created recently. Deployment experience has shown a cookie age of 2 minutes or less to be a reasonable limit.

If the user agent uses "Lax-allowing-unsafe" enforcement, it MUST apply the following modification to the retrieval algorithm defined in Section 5.6.3:

Replace the condition in the penultimate bullet point of step 1 of the retrieval algorithm reading

- \* The HTTP request associated with the retrieval uses a "safe" method.

with

- \* At least one of the following is true:
  1. The HTTP request associated with the retrieval uses a "safe" method.
  2. The cookie's same-site-flag is "Default" and the amount of time elapsed since the cookie's creation-time is at most a duration of the user agent's choosing.

## 5.5. Storage Model

The user agent stores the following fields about each cookie: name, value, expiry-time, domain, path, creation-time, last-access-time, persistent-flag, host-only-flag, secure-only-flag, http-only-flag, and same-site-flag.

When the user agent "receives a cookie" from a request-uri with name cookie-name, value cookie-value, and attributes cookie-attribute-list, the user agent MUST process the cookie as follows:

1. A user agent MAY ignore a received cookie in its entirety. See Section 5.3.

2. If cookie-name is empty and cookie-value is empty, abort these steps and ignore the cookie entirely.
3. If the cookie-name or the cookie-value contains a %x00-08 / %x0A-1F / %x7F character (CTL characters excluding HTAB), abort these steps and ignore the cookie entirely.
4. If the sum of the lengths of cookie-name and cookie-value is more than 4096 octets, abort these steps and ignore the cookie entirely.
5. Create a new cookie with name cookie-name, value cookie-value. Set the creation-time and the last-access-time to the current date and time.
6. If the cookie-attribute-list contains an attribute with an attribute-name of "Max-Age":
  1. Set the cookie's persistent-flag to true.
  2. Set the cookie's expiry-time to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Max-Age".

Otherwise, if the cookie-attribute-list contains an attribute with an attribute-name of "Expires" (and does not contain an attribute with an attribute-name of "Max-Age"):

1. Set the cookie's persistent-flag to true.
2. Set the cookie's expiry-time to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Expires".

Otherwise:

1. Set the cookie's persistent-flag to false.
  2. Set the cookie's expiry-time to the latest representable date.
7. If the cookie-attribute-list contains an attribute with an attribute-name of "Domain":
    1. Let the domain-attribute be the attribute-value of the last attribute in the cookie-attribute-list with both an attribute-name of "Domain" and an attribute-value whose length is no more than 1024 octets. (Note that a leading

%x2E ("."), if present, is ignored even though that character is not permitted, but a trailing %x2E ("."), if present, will cause the user agent to ignore the attribute.)

Otherwise:

1. Let the domain-attribute be the empty string.
8. If the domain-attribute contains a character that is not in the range of [USASCII] characters, abort these steps and ignore the cookie entirely.
9. If the user agent is configured to reject "public suffixes" and the domain-attribute is a public suffix:

1. If the domain-attribute is identical to the canonicalized request-host:

1. Let the domain-attribute be the empty string.

Otherwise:

1. Abort these steps and ignore the cookie entirely.

NOTE: This step prevents attacker.example from disrupting the integrity of site.example by setting a cookie with a Domain attribute of "example".

10. If the domain-attribute is non-empty:
  1. If the canonicalized request-host does not domain-match the domain-attribute:
    1. Abort these steps and ignore the cookie entirely.
  - Otherwise:
    1. Set the cookie's host-only-flag to false.
    2. Set the cookie's domain to the domain-attribute.

Otherwise:

1. Set the cookie's host-only-flag to true.
2. Set the cookie's domain to the canonicalized request-host.

11. If the cookie-attribute-list contains an attribute with an attribute-name of "Path", set the cookie's path to attribute-value of the last attribute in the cookie-attribute-list with both an attribute-name of "Path" and an attribute-value whose length is no more than 1024 octets. Otherwise, set the cookie's path to the default-path of the request-uri.
12. If the cookie-attribute-list contains an attribute with an attribute-name of "Secure", set the cookie's secure-only-flag to true. Otherwise, set the cookie's secure-only-flag to false.
13. If the scheme component of the request-uri does not denote a "secure" protocol (as defined by the user agent), and the cookie's secure-only-flag is true, then abort these steps and ignore the cookie entirely.
14. If the cookie-attribute-list contains an attribute with an attribute-name of "HttpOnly", set the cookie's http-only-flag to true. Otherwise, set the cookie's http-only-flag to false.
15. If the cookie was received from a "non-HTTP" API and the cookie's http-only-flag is true, abort these steps and ignore the cookie entirely.
16. If the cookie's secure-only-flag is false, and the scheme component of request-uri does not denote a "secure" protocol, then abort these steps and ignore the cookie entirely if the cookie store contains one or more cookies that meet all of the following criteria:
  1. Their name matches the name of the newly-created cookie.
  2. Their secure-only-flag is true.
  3. Their domain domain-matches the domain of the newly-created cookie, or vice-versa.
  4. The path of the newly-created cookie path-matches the path of the existing cookie.

Note: The path comparison is not symmetric, ensuring only that a newly-created, non-secure cookie does not overlay an existing secure cookie, providing some mitigation against cookie-fixing attacks. That is, given an existing secure cookie named 'a' with a path of '/login', a non-secure cookie named 'a' could be set for a path of '/' or '/foo', but not for a path of '/login' or '/login/en'.

17. If the cookie-attribute-list contains an attribute with an attribute-name of "SameSite", and an attribute-value of "Strict", "Lax", or "None", set the cookie's same-site-flag to the attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "SameSite". Otherwise, set the cookie's same-site-flag to "Default".
18. If the cookie's same-site-flag is not "None":
  1. If the cookie was received from a "non-HTTP" API, and the API was called from a browsing context's active document whose "site for cookies" is not same-site with the top-level origin, then abort these steps and ignore the newly created cookie entirely.
  2. If the cookie was received from a "same-site" request (as defined in Section 5.2), skip the remaining substeps and continue processing the cookie.
  3. If the cookie was received from a request which is navigating a top-level browsing context [HTML] (e.g. if the request's "reserved client" is either null or an environment whose "target browsing context" is a top-level browsing context), skip the remaining substeps and continue processing the cookie.

Note: Top-level navigations can create a cookie with any SameSite value, even if the new cookie wouldn't have been sent along with the request had it already existed prior to the navigation.
  4. Abort these steps and ignore the newly created cookie entirely.
19. If the cookie's "same-site-flag" is "None", abort these steps and ignore the cookie entirely unless the cookie's secure-only-flag is true.
20. If the cookie-name begins with a case-sensitive match for the string "\_\_Secure-", abort these steps and ignore the cookie entirely unless the cookie's secure-only-flag is true.
21. If the cookie-name begins with a case-sensitive match for the string "\_\_Host-", abort these steps and ignore the cookie entirely unless the cookie meets all the following criteria:
  1. The cookie's secure-only-flag is true.

2. The cookie's host-only-flag is true.
  3. The cookie-attribute-list contains an attribute with an attribute-name of "Path", and the cookie's path is /.
22. If the cookie store contains a cookie with the same name, domain, host-only-flag, and path as the newly-created cookie:
1. Let old-cookie be the existing cookie with the same name, domain, host-only-flag, and path as the newly-created cookie. (Notice that this algorithm maintains the invariant that there is at most one such cookie.)
  2. If the newly-created cookie was received from a "non-HTTP" API and the old-cookie's http-only-flag is true, abort these steps and ignore the newly created cookie entirely.
  3. Update the creation-time of the newly-created cookie to match the creation-time of the old-cookie.
  4. Remove the old-cookie from the cookie store.
23. Insert the newly-created cookie into the cookie store.

A cookie is "expired" if the cookie has an expiry date in the past.

The user agent MUST evict all expired cookies from the cookie store if, at any time, an expired cookie exists in the cookie store.

At any time, the user agent MAY "remove excess cookies" from the cookie store if the number of cookies sharing a domain field exceeds some implementation-defined upper bound (such as 50 cookies).

At any time, the user agent MAY "remove excess cookies" from the cookie store if the cookie store exceeds some predetermined upper bound (such as 3000 cookies).

When the user agent removes excess cookies from the cookie store, the user agent MUST evict cookies in the following priority order:

1. Expired cookies.
2. Cookies whose secure-only-flag is false, and which share a domain field with more than a predetermined number of other cookies.
3. Cookies that share a domain field with more than a predetermined number of other cookies.

#### 4. All cookies.

If two cookies have the same removal priority, the user agent MUST evict the cookie with the earliest last-access-time first.

When "the current session is over" (as defined by the user agent), the user agent MUST remove from the cookie store all cookies with the persistent-flag set to false.

#### 5.6. Retrieval Model

This section defines how cookies are retrieved from a cookie store in the form of a cookie-string. A "retrieval" is any event which requires generating a cookie-string. For example, a retrieval may occur in order to build a Cookie header field for an HTTP request, or may be required in order to return a cookie-string from a call to a "non-HTTP" API that provides access to cookies. A retrieval has an associated URI, same-site status, and type, which are defined below depending on the type of retrieval.

##### 5.6.1. The Cookie Header Field

The user agent includes stored cookies in the Cookie HTTP request header field.

When the user agent generates an HTTP request, the user agent MUST NOT attach more than one Cookie header field.

A user agent MAY omit the Cookie header field in its entirety. For example, the user agent might wish to block sending cookies during "third-party" requests from setting cookies (see Section 7.1).

If the user agent does attach a Cookie header field to an HTTP request, the user agent MUST compute the cookie-string following the algorithm defined in Section 5.6.3, where the retrieval's URI is the request-uri, the retrieval's same-site status is computed for the HTTP request as defined in Section 5.2, and the retrieval's type is "HTTP".

##### 5.6.2. Non-HTTP APIs

The user agent MAY implement "non-HTTP" APIs that can be used to access stored cookies.

A user agent MAY return an empty cookie-string in certain contexts, such as when a retrieval occurs within a third-party context (see Section 7.1).



If a user agent does return cookies for a given call to a "non-HTTP" API with an associated Document, then the user agent MUST compute the cookie-string following the algorithm defined in Section 5.6.3, where the retrieval's URI is defined by the caller (see [DOM-DOCUMENT-COOKIE]), the retrieval's same-site status is "same-site" if the Document's "site for cookies" is same-site with the top-level origin as defined in Section 5.2.1 (otherwise it is "cross-site"), and the retrieval's type is "non-HTTP".

### 5.6.3. Retrieval Algorithm

Given a cookie store and a retrieval, the following algorithm returns a cookie-string from a given cookie store.

1. Let cookie-list be the set of cookies from the cookie store that meets all of the following requirements:

- \* Either:

- The cookie's host-only-flag is true and the canonicalized host of the retrieval's URI is identical to the cookie's domain.

Or:

- The cookie's host-only-flag is false and the canonicalized host of the retrieval's URI domain-matches the cookie's domain.

- \* The retrieval's URI's path path-matches the cookie's path.

- \* If the cookie's secure-only-flag is true, then the retrieval's URI's scheme must denote a "secure" protocol (as defined by the user agent).

NOTE: The notion of a "secure" protocol is not defined by this document. Typically, user agents consider a protocol secure if the protocol makes use of transport-layer security, such as SSL or TLS. For example, most user agents consider "https" to be a scheme that denotes a secure protocol.

- \* If the cookie's http-only-flag is true, then exclude the cookie if the retrieval's type is "non-HTTP".
- \* If the cookie's same-site-flag is not "None" and the retrieval's same-site status is "cross-site", then exclude the cookie unless all of the following conditions are met:

- The retrieval's type is "HTTP".
  - The same-site-flag is "Lax" or "Default".
  - The HTTP request associated with the retrieval uses a "safe" method.
  - The target browsing context of the HTTP request associated with the retrieval is a top-level browsing context.
2. The user agent SHOULD sort the cookie-list in the following order:
- \* Cookies with longer paths are listed before cookies with shorter paths.
  - \* Among cookies that have equal-length path fields, cookies with earlier creation-times are listed before cookies with later creation-times.
- NOTE: Not all user agents sort the cookie-list in this order, but this order reflects common practice when this document was written, and, historically, there have been servers that (erroneously) depended on this order.
3. Update the last-access-time of each cookie in the cookie-list to the current date and time.
4. Serialize the cookie-list into a cookie-string by processing each cookie in the cookie-list in order:
- 1. If the cookies' name is not empty, output the cookie's name followed by the %x3D ("=") character.
  - 2. If the cookies' value is not empty, output the cookie's value.
  - 3. If there is an unprocessed cookie in the cookie-list, output the characters %x3B and %x20 ("; ").

NOTE: Despite its name, the cookie-string is actually a sequence of octets, not a sequence of characters. To convert the cookie-string (or components thereof) into a sequence of characters (e.g., for presentation to the user), the user agent might wish to try using the UTF-8 character encoding [RFC3629] to decode the octet sequence. This decoding might fail, however, because not every sequence of octets is valid UTF-8.

## 6. Implementation Considerations

### 6.1. Limits

Practical user agent implementations have limits on the number and size of cookies that they can store. General-use user agents SHOULD provide each of the following minimum capabilities:

- \* At least 50 cookies per domain.
- \* At least 3000 cookies total.

User agents MAY limit the maximum number of cookies they store, and may evict any cookie at any time (whether at the request of the user or due to implementation limitations).

Note that a limit on the maximum number of cookies also limits the total size of the stored cookies, due to the length limits which MUST be enforced in Section 5.4.

Servers SHOULD use as few and as small cookies as possible to avoid reaching these implementation limits and to minimize network bandwidth due to the Cookie header field being included in every request.

Servers SHOULD gracefully degrade if the user agent fails to return one or more cookies in the Cookie header field because the user agent might evict any cookie at any time.

### 6.2. Application Programming Interfaces

One reason the Cookie and Set-Cookie header fields use such esoteric syntax is that many platforms (both in servers and user agents) provide a string-based application programming interface (API) to cookies, requiring application-layer programmers to generate and parse the syntax used by the Cookie and Set-Cookie header fields, which many programmers have done incorrectly, resulting in interoperability problems.

Instead of providing string-based APIs to cookies, platforms would be well-served by providing more semantic APIs. It is beyond the scope of this document to recommend specific API designs, but there are clear benefits to accepting an abstract "Date" object instead of a serialized date string.

### 6.3. IDNA Dependency and Migration

IDNA2008 [RFC5890] supersedes IDNA2003 [RFC3490]. However, there are differences between the two specifications, and thus there can be differences in processing (e.g., converting) domain name labels that have been registered under one from those registered under the other. There will be a transition period of some time during which IDNA2003-based domain name labels will exist in the wild. User agents SHOULD implement IDNA2008 [RFC5890] and MAY implement [UTS46] or [RFC5895] in order to facilitate their IDNA transition. If a user agent does not implement IDNA2008, the user agent MUST implement IDNA2003 [RFC3490].

## 7. Privacy Considerations

Cookies' primary privacy risk is their ability to correlate user activity. This can happen on a single site, but is most problematic when activity is tracked across different, seemingly unconnected Web sites to build a user profile.

Over time, this capability (warned against explicitly in [RFC2109] and all of its successors) has become widely used for varied reasons including:

- \* authenticating users across sites,
- \* assembling information on users,
- \* protecting against fraud and other forms of undesirable traffic,
- \* targeting advertisements at specific users or at users with specified attributes,
- \* measuring how often ads are shown to users, and
- \* recognizing when an ad resulted in a change in user behavior.

While not every use of cookies is necessarily problematic for privacy, their potential for abuse has become a widespread concern in the Internet community and broader society. In response to these concerns, user agents have actively constrained cookie functionality in various ways (as allowed and encouraged by previous specifications), while avoiding disruption to features they judge desirable for the health of the Web.

It is too early to declare consensus on which specific mechanism(s) should be used to mitigate cookies' privacy impact; user agents' ongoing changes to how they are handled are best characterised as experiments that can provide input into that eventual consensus.

Instead, this document describes limited, general mitigations against the privacy risks associated with cookies that enjoy wide deployment at the time of writing. It is expected that implementations will continue to experiment and impose stricter, more well-defined limitations on cookies over time. Future versions of this document might codify those mechanisms based upon deployment experience. If functions that currently rely on cookies can be supported by separate, targeted mechanisms, they might be documented in separate specifications and stricter limitations on cookies might become feasible.

Note that cookies are not the only mechanism that can be used to track users across sites, so while these mitigations are necessary to improve Web privacy, they are not sufficient on their own.

### 7.1. Third-Party Cookies

A "third-party" or cross-site cookie is one that is associated with embedded content (such as scripts, images, stylesheets, frames) that is obtained from a different server than the one that hosts the primary resource (usually, the Web page that the user is viewing). Third-party cookies are often used to correlate users' activity on different sites.

Because of their inherent privacy issues, most user agents now limit third-party cookies in a variety of ways. Some completely block third-party cookies by refusing to process third-party Set-Cookie header fields and refusing to send third-party Cookie header fields. Some partition cookies based upon the first-party context, so that different cookies are sent depending on the site being browsed. Some block cookies based upon user agent cookie policy and/or user controls.

While this document does not endorse or require a specific approach, it is RECOMMENDED that user agents adopt a policy for third-party cookies that is as restrictive as compatibility constraints permit. Consequently, resources cannot rely upon third-party cookies being treated consistently by user agents for the foreseeable future.

### 7.2. Cookie Policy

User agents MAY enforce a cookie policy consisting of restrictions on how cookies may be used or ignored (see Section 5.3).

A cookie policy may govern which domains or parties, as in first and third parties (see Section 7.1), for which the user agent will allow cookie access. The policy can also define limits on cookie size, cookie expiry (see Section 4.1.2.1 and Section 4.1.2.2), and the number of cookies per domain or in total.

The recommended cookie expiry upper limit is 400 days. User agents may set a lower limit to enforce shorter data retention timelines, or set the limit higher to support longer retention when appropriate (e.g., server-to-server communication over HTTPS).

The goal of a restrictive cookie policy is often to improve security or privacy. User agents often allow users to change the cookie policy (see Section 7.3).

### 7.3. User Controls

User agents SHOULD provide users with a mechanism for managing the cookies stored in the cookie store. For example, a user agent might let users delete all cookies received during a specified time period or all the cookies related to a particular domain. In addition, many user agents include a user interface element that lets users examine the cookies stored in their cookie store.

User agents SHOULD provide users with a mechanism for disabling cookies. When cookies are disabled, the user agent MUST NOT include a Cookie header field in outbound HTTP requests and the user agent MUST NOT process Set-Cookie header fields in inbound HTTP responses.

User agents MAY offer a way to change the cookie policy (see Section 7.2).

User agents MAY provide users the option of preventing persistent storage of cookies across sessions. When configured thusly, user agents MUST treat all received cookies as if the persistent-flag were set to false. Some popular user agents expose this functionality via "private browsing" mode [Aggarwal2010].

### 7.4. Expiration Dates

Although servers can set the expiration date for cookies to the distant future, most user agents do not actually retain cookies for multiple decades. Rather than choosing gratuitously long expiration periods, servers SHOULD promote user privacy by selecting reasonable cookie expiration periods based on the purpose of the cookie. For example, a typical session identifier might reasonably be set to expire in two weeks.

## 8. Security Considerations

### 8.1. Overview

Cookies have a number of security pitfalls. This section overviews a few of the more salient issues.

In particular, cookies encourage developers to rely on ambient authority for authentication, often becoming vulnerable to attacks such as cross-site request forgery [CSRF]. Also, when storing session identifiers in cookies, developers often create session fixation vulnerabilities.

Transport-layer encryption, such as that employed in HTTPS, is insufficient to prevent a network attacker from obtaining or altering a victim's cookies because the cookie protocol itself has various vulnerabilities (see "Weak Confidentiality" and "Weak Integrity", below). In addition, by default, cookies do not provide confidentiality or integrity from network attackers, even when used in conjunction with HTTPS.

### 8.2. Ambient Authority

A server that uses cookies to authenticate users can suffer security vulnerabilities because some user agents let remote parties issue HTTP requests from the user agent (e.g., via HTTP redirects or HTML forms). When issuing those requests, user agents attach cookies even if the remote party does not know the contents of the cookies, potentially letting the remote party exercise authority at an unwary server.

Although this security concern goes by a number of names (e.g., cross-site request forgery, confused deputy), the issue stems from cookies being a form of ambient authority. Cookies encourage server operators to separate designation (in the form of URLs) from authorization (in the form of cookies). Consequently, the user agent might supply the authorization for a resource designated by the attacker, possibly causing the server or its clients to undertake actions designated by the attacker as though they were authorized by the user.

Instead of using cookies for authorization, server operators might wish to consider entangling designation and authorization by treating URLs as capabilities. Instead of storing secrets in cookies, this approach stores secrets in URLs, requiring the remote entity to supply the secret itself. Although this approach is not a panacea, judicious application of these principles can lead to more robust security.

### 8.3. Clear Text

Unless sent over a secure channel (such as TLS), the information in the Cookie and Set-Cookie header fields is transmitted in the clear.

1. All sensitive information conveyed in these header fields is exposed to an eavesdropper.
2. A malicious intermediary could alter the header fields as they travel in either direction, with unpredictable results.
3. A malicious client could alter the Cookie header fields before transmission, with unpredictable results.

Servers SHOULD encrypt and sign the contents of cookies (using whatever format the server desires) when transmitting them to the user agent (even when sending the cookies over a secure channel). However, encrypting and signing cookie contents does not prevent an attacker from transplanting a cookie from one user agent to another or from replaying the cookie at a later time.

In addition to encrypting and signing the contents of every cookie, servers that require a higher level of security SHOULD use the Cookie and Set-Cookie header fields only over a secure channel. When using cookies over a secure channel, servers SHOULD set the Secure attribute (see Section 4.1.2.5) for every cookie. If a server does not set the Secure attribute, the protection provided by the secure channel will be largely moot.

For example, consider a webmail server that stores a session identifier in a cookie and is typically accessed over HTTPS. If the server does not set the Secure attribute on its cookies, an active network attacker can intercept any outbound HTTP request from the user agent and redirect that request to the webmail server over HTTP. Even if the webmail server is not listening for HTTP connections, the user agent will still include cookies in the request. The active network attacker can intercept these cookies, replay them against the server, and learn the contents of the user's email. If, instead, the server had set the Secure attribute on its cookies, the user agent would not have included the cookies in the clear-text request.

### 8.4. Session Identifiers

Instead of storing session information directly in a cookie (where it might be exposed to or replayed by an attacker), servers commonly store a nonce (or "session identifier") in a cookie. When the server receives an HTTP request with a nonce, the server can look up state information associated with the cookie using the nonce as a key.



Using session identifier cookies limits the damage an attacker can cause if the attacker learns the contents of a cookie because the nonce is useful only for interacting with the server (unlike non-nonce cookie content, which might itself be sensitive). Furthermore, using a single nonce prevents an attacker from "splicing" together cookie content from two interactions with the server, which could cause the server to behave unexpectedly.

Using session identifiers is not without risk. For example, the server SHOULD take care to avoid "session fixation" vulnerabilities. A session fixation attack proceeds in three steps. First, the attacker transplants a session identifier from his or her user agent to the victim's user agent. Second, the victim uses that session identifier to interact with the server, possibly imbuing the session identifier with the user's credentials or confidential information. Third, the attacker uses the session identifier to interact with server directly, possibly obtaining the user's authority or confidential information.

#### 8.5. Weak Confidentiality

Cookies do not provide isolation by port. If a cookie is readable by a service running on one port, the cookie is also readable by a service running on another port of the same server. If a cookie is writable by a service on one port, the cookie is also writable by a service running on another port of the same server. For this reason, servers SHOULD NOT both run mutually distrusting services on different ports of the same host and use cookies to store security-sensitive information.

Cookies do not provide isolation by scheme. Although most commonly used with the http and https schemes, the cookies for a given host might also be available to other schemes, such as ftp and gopher. Although this lack of isolation by scheme is most apparent in non-HTTP APIs that permit access to cookies (e.g., HTML's document.cookie API), the lack of isolation by scheme is actually present in requirements for processing cookies themselves (e.g., consider retrieving a URI with the gopher scheme via HTTP).

Cookies do not always provide isolation by path. Although the network-level protocol does not send cookies stored for one path to another, some user agents expose cookies via non-HTTP APIs, such as HTML's document.cookie API. Because some of these user agents (e.g., web browsers) do not isolate resources received from different paths, a resource retrieved from one path might be able to access cookies stored for another path.

## 8.6. Weak Integrity

Cookies do not provide integrity guarantees for sibling domains (and their subdomains). For example, consider `foo.site.example` and `bar.site.example`. The `foo.site.example` server can set a cookie with a Domain attribute of `"site.example"` (possibly overwriting an existing `"site.example"` cookie set by `bar.site.example`), and the user agent will include that cookie in HTTP requests to `bar.site.example`. In the worst case, `bar.site.example` will be unable to distinguish this cookie from a cookie it set itself. The `foo.site.example` server might be able to leverage this ability to mount an attack against `bar.site.example`.

Even though the Set-Cookie header field supports the Path attribute, the Path attribute does not provide any integrity protection because the user agent will accept an arbitrary Path attribute in a Set-Cookie header field. For example, an HTTP response to a request for `http://site.example/foo/bar` can set a cookie with a Path attribute of `"/qux"`. Consequently, servers SHOULD NOT both run mutually distrusting services on different paths of the same host and use cookies to store security-sensitive information.

An active network attacker can also inject cookies into the Cookie header field sent to `https://site.example/` by impersonating a response from `http://site.example/` and injecting a Set-Cookie header field. The HTTPS server at `site.example` will be unable to distinguish these cookies from cookies that it set itself in an HTTPS response. An active network attacker might be able to leverage this ability to mount an attack against `site.example` even if `site.example` uses HTTPS exclusively.

Servers can partially mitigate these attacks by encrypting and signing the contents of their cookies, or by naming the cookie with the `__Secure-` prefix. However, using cryptography does not mitigate the issue completely because an attacker can replay a cookie he or she received from the authentic `site.example` server in the user's session, with unpredictable results.

Finally, an attacker might be able to force the user agent to delete cookies by storing a large number of cookies. Once the user agent reaches its storage limit, the user agent will be forced to evict some cookies. Servers SHOULD NOT rely upon user agents retaining cookies.

### 8.7. Reliance on DNS

Cookies rely upon the Domain Name System (DNS) for security. If the DNS is partially or fully compromised, the cookie protocol might fail to provide the security properties required by applications.

### 8.8. SameSite Cookies

#### 8.8.1. Defense in depth

"SameSite" cookies offer a robust defense against CSRF attack when deployed in strict mode, and when supported by the client. It is, however, prudent to ensure that this designation is not the extent of a site's defense against CSRF, as same-site navigations and submissions can certainly be executed in conjunction with other attack vectors such as cross-site scripting.

Developers are strongly encouraged to deploy the usual server-side defenses (CSRF tokens, ensuring that "safe" HTTP methods are idempotent, etc) to mitigate the risk more fully.

Additionally, client-side techniques such as those described in [app-isolation] may also prove effective against CSRF, and are certainly worth exploring in combination with "SameSite" cookies.

#### 8.8.2. Top-level Navigations

Setting the SameSite attribute in "strict" mode provides robust defense in depth against CSRF attacks, but has the potential to confuse users unless sites' developers carefully ensure that their cookie-based session management systems deal reasonably well with top-level navigations.

Consider the scenario in which a user reads their email at MegaCorp Inc's webmail provider <https://site.example/>. They might expect that clicking on an emailed link to <https://projects.example/secret/> project would show them the secret project that they're authorized to see, but if [https://projects.example](https://projects.example/) has marked their session cookies as SameSite=Strict, then this cross-site navigation won't send them along with the request. [https://projects.example](https://projects.example/) will render a 404 error to avoid leaking secret information, and the user will be quite confused.

Developers can avoid this confusion by adopting a session management system that relies on not one, but two cookies: one conceptually granting "read" access, another granting "write" access. The latter could be marked as SameSite=Strict, and its absence would prompt a reauthentication step before executing any non-idempotent action.

The former could be marked as `SameSite=Lax`, in order to allow users access to data via top-level navigation, or `SameSite=None`, to permit access in all contexts (including cross-site embedded contexts).

#### 8.8.3. Mashups and Widgets

The `Lax` and `Strict` values for the `SameSite` attribute are inappropriate for some important use-cases. In particular, note that content intended for embedding in cross-site contexts (social networking widgets or commenting services, for instance) will not have access to same-site cookies. Cookies which are required in these situations should be marked with `SameSite=None` to allow access in cross-site contexts.

Likewise, some forms of Single-Sign-On might require cookie-based authentication in a cross-site context; these mechanisms will not function as intended with same-site cookies and will also require `SameSite=None`.

#### 8.8.4. Server-controlled

`SameSite` cookies in and of themselves don't do anything to address the general privacy concerns outlined in Section 7.1 of [RFC6265]. The "`SameSite`" attribute is set by the server, and serves to mitigate the risk of certain kinds of attacks that the server is worried about. The user is not involved in this decision. Moreover, a number of side-channels exist which could allow a server to link distinct requests even in the absence of cookies (for example, connection and/or socket pooling between same-site and cross-site requests).

#### 8.8.5. Reload navigations

Requests issued for reloads triggered through user interface elements (such as a refresh button on a toolbar) are same-site only if the reloaded document was originally navigated to via a same-site request. This differs from the handling of other reload navigations, which are always same-site if top-level, since the source browsing context's active document is precisely the document being reloaded.

This special handling of reloads triggered through a user interface element avoids sending SameSite cookies on user-initiated reloads if they were withheld on the original navigation (i.e., if the initial navigation were cross-site). If the reload navigation were instead considered same-site, and sent all the initially withheld SameSite cookies, the security benefits of withholding the cookies in the first place would be nullified. This is especially important given that the absence of SameSite cookies withheld on a cross-site navigation request may lead to visible site breakage, prompting the user to trigger a reload.

For example, suppose the user clicks on a link from `https://attacker.example/` to `https://victim.example/`. This is a cross-site request, so `SameSite=Strict` cookies are withheld. Suppose this causes `https://victim.example/` to appear broken, because the site only displays its sensitive content if a particular SameSite cookie is present in the request. The user, frustrated by the unexpectedly broken site, presses refresh on their browser's toolbar. To now consider the reload request same-site and send the initially withheld SameSite cookie would defeat the purpose of withholding it in the first place, as the reload navigation triggered through the user interface may replay the original (potentially malicious) request. Thus, the reload request should be considered cross-site, like the request that initially navigated to the page.

#### 8.8.6. Top-level requests with "unsafe" methods

The "Lax" enforcement mode described in Section 5.4.7.1 allows a cookie to be sent with a cross-site HTTP request if and only if it is a top-level navigation with a "safe" HTTP method. Implementation experience shows that this is difficult to apply as the default behavior, as some sites may rely on cookies not explicitly specifying a SameSite attribute being included on top-level cross-site requests with "unsafe" HTTP methods (as was the case prior to the introduction of the SameSite attribute).

For example, a login flow may involve a cross-site top-level POST request to an endpoint which expects a cookie with login information. For such a cookie, "Lax" enforcement is not appropriate, as it would cause the cookie to be excluded due to the unsafe HTTP request method. On the other hand, "None" enforcement would allow the cookie to be sent with all cross-site requests, which may not be desirable due to the cookie's sensitive contents.

The "Lax-allowing-unsafe" enforcement mode described in Section 5.4.7.2 retains some of the protections of "Lax" enforcement (as compared to "None") while still allowing cookies to be sent cross-site with unsafe top-level requests.

As a more permissive variant of "Lax" mode, "Lax-allowing-unsafe" mode necessarily provides fewer protections against CSRF. Ultimately, the provision of such an enforcement mode should be seen as a temporary, transitional measure to ease adoption of "Lax" enforcement by default.

## 9. IANA Considerations

### 9.1. Cookie

The permanent message header field registry (see [RFC3864]) needs to be updated with the following registration:

Header field name: Cookie

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 5.6.1)

### 9.2. Set-Cookie

The permanent message header field registry (see [RFC3864]) needs to be updated with the following registration:

Header field name: Set-Cookie

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document: this specification (Section 5.4)

### 9.3. Cookie Attribute Registry

IANA is requested to create the "Cookie Attribute Registry", defining the name space of attribute used to control cookies' behavior. The registry should be maintained at <https://www.iana.org/assignments/cookie-attribute-names> (<https://www.iana.org/assignments/cookie-attribute-names>).

### 9.3.1. Procedure

Each registered attribute name is associated with a description, and a reference detailing how the attribute is to be processed and stored.

New registrations happen on a "RFC Required" basis (see Section 4.7 of [RFC8126]). The attribute to be registered MUST match the extension-av syntax defined in Section 4.1.1. Note that attribute names are generally defined in CamelCase, but technically accepted case-insensitively.

### 9.3.2. Registration

The "Cookie Attribute Registry" should be created with the registrations below:

Name	Reference
Domain	Section 4.1.2.3 of this document
Expires	Section 4.1.2.1 of this document
HttpOnly	Section 4.1.2.6 of this document
Max-Age	Section 4.1.2.2 of this document
Path	Section 4.1.2.4 of this document
SameSite	Section 4.1.2.7 of this document
Secure	Section 4.1.2.5 of this document

Table 1

## 10. References

### 10.1. Normative References

- [DOM-DOCUMENT-COOKIE] WHATWG, "HTML - Living Standard", 18 May 2021, <<https://html.spec.whatwg.org/#dom-document-cookie>>.
- [FETCH] van Kesteren, A., "Fetch", n.d., <<https://fetch.spec.whatwg.org/>>.

- [HTML] Hickson, I., Pieters, S., van Kesteren, A., Jägenstedt, P., and D. Denicola, "HTML", n.d., <<https://html.spec.whatwg.org/>>.
- [HTTPSEM] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-19, 12 September 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-19>>.
- [RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <<https://www.rfc-editor.org/rfc/rfc1034>>.
- [RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <<https://www.rfc-editor.org/rfc/rfc1123>>.
- [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/rfc/rfc2119>>.
- [RFC3490] Costello, A., "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003, <<https://www.rfc-editor.org/rfc/rfc3490>>. See Section 6.3 for an explanation why the normative reference to an obsoleted specification is needed.
- [RFC4790] Newman, C., Duerst, M., and A. Gulbrandsen, "Internet Application Protocol Collation Registry", RFC 4790, DOI 10.17487/RFC4790, March 2007, <<https://www.rfc-editor.org/rfc/rfc4790>>.
- [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/rfc/rfc5234>>.
- [RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <<https://www.rfc-editor.org/rfc/rfc5890>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/rfc/rfc6454>>.



[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/rfc/rfc8126>>.

[SAMESITE] WHATWG, "HTML - Living Standard", 26 January 2021, <<https://html.spec.whatwg.org/#same-site>>.

[SERVICE-WORKERS] Russell, A., Song, J., and J. Archibald, "Service Workers", n.d., <<http://www.w3.org/TR/service-workers/>>.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

## 10.2. Informative References

[Aggarwal2010] Aggarwal, G., Burzstein, E., Jackson, C., and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers", 2010, <[http://www.usenix.org/events/sec10/tech/full\\_papers/Aggarwal.pdf](http://www.usenix.org/events/sec10/tech/full_papers/Aggarwal.pdf)>.

[app-isolation] Chen, E., Bau, J., Reis, C., Barth, A., and C. Jackson, "App Isolation - Get the Security of Multiple Browsers with Just One", 2011, <<http://www.collinjackson.com/research/papers/appisolation.pdf>>.

[CSRF] Barth, A., Jackson, C., and J. Mitchell, "Robust Defenses for Cross-Site Request Forgery", DOI 10.1145/1455770.1455782, ISBN 978-1-59593-810-7, ACM CCS '08: Proceedings of the 15th ACM conference on Computer and communications security (pages 75-88), October 2008, <<http://portal.acm.org/citation.cfm?id=1455770.1455782>>.

[I-D.ietf-httpbis-cookie-alone] West, M., "Deprecate modification of 'secure' cookies from non-secure origins", Work in Progress, Internet-Draft, draft-ietf-httpbis-cookie-alone-01, 5 September 2016, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cookie-alone-01>>.

- [I-D.ietf-httpbis-cookie-prefixes]  
West, M., "Cookie Prefixes", Work in Progress, Internet-Draft, draft-ietf-httpbis-cookie-prefixes-00, 23 February 2016, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cookie-prefixes-00>>.
- [I-D.ietf-httpbis-cookie-same-site]  
West, M. and M. Goodwin, "Same-Site Cookies", Work in Progress, Internet-Draft, draft-ietf-httpbis-cookie-same-site-00, 20 June 2016, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cookie-same-site-00>>.
- [prerendering]  
Bentzel, C., "Chrome Prerendering", n.d., <<https://www.chromium.org/developers/design-documents/prerender>>.
- [PSL] "Public Suffix List", n.d., <<https://publicsuffix.org/list/>>.
- [RFC2109] Kristol, D. and L. Montulli, "HTTP State Management Mechanism", RFC 2109, DOI 10.17487/RFC2109, February 1997, <<https://www.rfc-editor.org/rfc/rfc2109>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<https://www.rfc-editor.org/rfc/rfc2818>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/rfc/rfc3629>>.
- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <<https://www.rfc-editor.org/rfc/rfc3864>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/rfc/rfc3986>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/rfc/rfc4648>>.

- [RFC5895] Resnick, P. and P. Hoffman, "Mapping Characters for Internationalized Domain Names in Applications (IDNA) 2008", RFC 5895, DOI 10.17487/RFC5895, September 2010, <<https://www.rfc-editor.org/rfc/rfc5895>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.
- [RFC7034] Ross, D. and T. Gondrom, "HTTP Header Field X-Frame-Options", RFC 7034, DOI 10.17487/RFC7034, October 2013, <<https://www.rfc-editor.org/rfc/rfc7034>>.
- [UTS46] Davis, M. and M. Suignard, "Unicode IDNA Compatibility Processing", UNICODE Unicode Technical Standards # 46, June 2016, <<http://unicode.org/reports/tr46/>>.

## Appendix A. Changes

### A.1. draft-ietf-httpbis-rfc6265bis-00

- \* Port [RFC6265] to Markdown. No (intentional) normative changes.

### A.2. draft-ietf-httpbis-rfc6265bis-01

- \* Fixes to formatting caused by mistakes in the initial port to Markdown:
  - <https://github.com/httpwg/http-extensions/issues/243>  
(<https://github.com/httpwg/http-extensions/issues/243>)
  - <https://github.com/httpwg/http-extensions/issues/246>  
(<https://github.com/httpwg/http-extensions/issues/246>)
- \* Addresses errata 3444 by updating the path-value and extension-av grammar, errata 4148 by updating the day-of-month, year, and time grammar, and errata 3663 by adding the requested note.  
[https://www.rfc-editor.org/errata\\_search.php?rfc=6265](https://www.rfc-editor.org/errata_search.php?rfc=6265)  
([https://www.rfc-editor.org/errata\\_search.php?rfc=6265](https://www.rfc-editor.org/errata_search.php?rfc=6265))
- \* Dropped Cookie2 and Set-Cookie2 from the IANA Considerations section: <https://github.com/httpwg/http-extensions/issues/247>  
(<https://github.com/httpwg/http-extensions/issues/247>)
- \* Merged the recommendations from [I-D.ietf-httpbis-cookie-alone], removing the ability for a non-secure origin to set cookies with a 'secure' flag, and to overwrite cookies whose 'secure' flag is true.

- \* Merged the recommendations from [I-D.ietf-httpbis-cookie-prefixes], adding `__Secure-` and `__Host-` cookie name prefix processing instructions.

#### A.3. draft-ietf-httpbis-rfc6265bis-02

- \* Merged the recommendations from [I-D.ietf-httpbis-cookie-same-site], adding support for the `SameSite` attribute.
- \* Closed a number of editorial bugs:
  - Clarified address bar behavior for `SameSite` cookies:  
<https://github.com/httpwg/http-extensions/issues/201>  
(<https://github.com/httpwg/http-extensions/issues/201>)
  - Added the word "Cookies" to the document's name:  
<https://github.com/httpwg/http-extensions/issues/204>  
(<https://github.com/httpwg/http-extensions/issues/204>)
  - Clarified that the `__Host-` prefix requires an explicit `Path` attribute: <https://github.com/httpwg/http-extensions/issues/222>  
(<https://github.com/httpwg/http-extensions/issues/222>)
  - Expanded the options for dealing with third-party cookies to include a brief mention of partitioning based on first-party:  
<https://github.com/httpwg/http-extensions/issues/248>  
(<https://github.com/httpwg/http-extensions/issues/248>)
  - Noted that double-quotes in cookie values are part of the value, and are not stripped: <https://github.com/httpwg/http-extensions/issues/295> (<https://github.com/httpwg/http-extensions/issues/295>)
  - Fixed the "site for cookies" algorithm to return something that makes sense: <https://github.com/httpwg/http-extensions/issues/302> (<https://github.com/httpwg/http-extensions/issues/302>)

#### A.4. draft-ietf-httpbis-rfc6265bis-03

- \* Clarified handling of invalid `SameSite` values:  
<https://github.com/httpwg/http-extensions/issues/389>  
(<https://github.com/httpwg/http-extensions/issues/389>)

- \* Reflect widespread implementation practice of including a cookie's host-only-flag when calculating its uniqueness:  
<https://github.com/httpwg/http-extensions/issues/199>  
(<https://github.com/httpwg/http-extensions/issues/199>)
- \* Introduced an explicit "None" value for the SameSite attribute:  
<https://github.com/httpwg/http-extensions/issues/788>  
(<https://github.com/httpwg/http-extensions/issues/788>)

#### A.5. draft-ietf-httpbis-rfc6265bis-04

- \* Allow SameSite cookies to be set for all top-level navigations.  
<https://github.com/httpwg/http-extensions/issues/594>  
(<https://github.com/httpwg/http-extensions/issues/594>)
- \* Treat Set-Cookie: token as creating the cookie ("", "token"):  
<https://github.com/httpwg/http-extensions/issues/159>  
(<https://github.com/httpwg/http-extensions/issues/159>)
- \* Reject cookies with neither name nor value (e.g. Set-Cookie: = and Set-Cookie:: <https://github.com/httpwg/http-extensions/issues/159> (<https://github.com/httpwg/http-extensions/issues/159>))
- \* Clarified behavior of multiple SameSite attributes in a cookie string: <https://github.com/httpwg/http-extensions/issues/901>  
(<https://github.com/httpwg/http-extensions/issues/901>)

#### A.6. draft-ietf-httpbis-rfc6265bis-05

- \* Typos and editorial fixes: <https://github.com/httpwg/http-extensions/pull/1035> (<https://github.com/httpwg/http-extensions/pull/1035>), <https://github.com/httpwg/http-extensions/pull/1038> (<https://github.com/httpwg/http-extensions/pull/1038>), <https://github.com/httpwg/http-extensions/pull/1040> (<https://github.com/httpwg/http-extensions/pull/1040>), <https://github.com/httpwg/http-extensions/pull/1047> (<https://github.com/httpwg/http-extensions/pull/1047>).

#### A.7. draft-ietf-httpbis-rfc6265bis-06

- \* Editorial fixes: <https://github.com/httpwg/http-extensions/issues/1059> (<https://github.com/httpwg/http-extensions/issues/1059>), <https://github.com/httpwg/http-extensions/issues/1158> (<https://github.com/httpwg/http-extensions/issues/1158>).

- \* Created a registry for cookie attribute names:  
<https://github.com/httpwg/http-extensions/pull/1060>  
(<https://github.com/httpwg/http-extensions/pull/1060>).
- \* Tweaks to ABNF for cookie-pair and the Cookie header production:  
<https://github.com/httpwg/http-extensions/issues/1074>  
(<https://github.com/httpwg/http-extensions/issues/1074>),  
<https://github.com/httpwg/http-extensions/issues/1119>  
(<https://github.com/httpwg/http-extensions/issues/1119>).
- \* Fixed serialization for nameless/valueless cookies:  
<https://github.com/httpwg/http-extensions/pull/1143>  
(<https://github.com/httpwg/http-extensions/pull/1143>).
- \* Converted a normative reference to Mozilla's Public Suffix List [PSL] into an informative reference: <https://github.com/httpwg/http-extensions/issues/1159> (<https://github.com/httpwg/http-extensions/issues/1159>).

#### A.8. draft-ietf-httpbis-rfc6265bis-07

- \* Moved instruction to ignore cookies with empty cookie-name and cookie-value from Section 5.4 to Section 5.5 to ensure that they apply to cookies created without parsing a cookie string:  
<https://github.com/httpwg/http-extensions/issues/1234>  
(<https://github.com/httpwg/http-extensions/issues/1234>).
- \* Add a default enforcement value to the same-site-flag, equivalent to "SameSite=Lax": <https://github.com/httpwg/http-extensions/pull/1325> (<https://github.com/httpwg/http-extensions/pull/1325>).
- \* Require a Secure attribute for "SameSite=None":  
<https://github.com/httpwg/http-extensions/pull/1323>  
(<https://github.com/httpwg/http-extensions/pull/1323>).
- \* Consider scheme when running the same-site algorithm:  
<https://github.com/httpwg/http-extensions/pull/1324>  
(<https://github.com/httpwg/http-extensions/pull/1324>).

#### A.9. draft-ietf-httpbis-rfc6265bis-08

- \* Define "same-site" for reload navigation requests, e.g. those triggered via user interface elements: <https://github.com/httpwg/http-extensions/pull/1384> (<https://github.com/httpwg/http-extensions/pull/1384>)

- \* Consider redirects when defining same-site:  
<https://github.com/httpwg/http-extensions/pull/1348>  
(<https://github.com/httpwg/http-extensions/pull/1348>)
- \* Align on using HTML terminology for origins:  
<https://github.com/httpwg/http-extensions/pull/1416>  
(<https://github.com/httpwg/http-extensions/pull/1416>)
- \* Modify cookie parsing and creation algorithms in Section 5.4 and Section 5.5 to explicitly handle control characters:  
<https://github.com/httpwg/http-extensions/pull/1420>  
(<https://github.com/httpwg/http-extensions/pull/1420>)
- \* Refactor cookie retrieval algorithm to support non-HTTP APIs:  
<https://github.com/httpwg/http-extensions/pull/1428>  
(<https://github.com/httpwg/http-extensions/pull/1428>)
- \* Define "Lax-allowing-unsafe" SameSite enforcement mode:  
<https://github.com/httpwg/http-extensions/pull/1435>  
(<https://github.com/httpwg/http-extensions/pull/1435>)
- \* Consistently use "header field" (vs 'header'):  
<https://github.com/httpwg/http-extensions/pull/1527>  
(<https://github.com/httpwg/http-extensions/pull/1527>)

#### A.10. draft-ietf-httpbis-rfc6265bis-09

- \* Update cookie size requirements: <https://github.com/httpwg/http-extensions/pull/1563> (<https://github.com/httpwg/http-extensions/pull/1563>)
- \* Reject cookies with control characters: <https://github.com/httpwg/http-extensions/pull/1576> (<https://github.com/httpwg/http-extensions/pull/1576>)
- \* No longer treat horizontal tab as a control character:  
<https://github.com/httpwg/http-extensions/pull/1589>  
(<https://github.com/httpwg/http-extensions/pull/1589>)
- \* Specify empty domain attribute handling:  
<https://github.com/httpwg/http-extensions/pull/1709>  
(<https://github.com/httpwg/http-extensions/pull/1709>)

#### A.11. draft-ietf-httpbis-rfc6265bis-10

- \* Standardize Max-Age/Expires upper bound:  
<https://github.com/httpwg/http-extensions/pull/1732>  
(<https://github.com/httpwg/http-extensions/pull/1732>)

## Acknowledgements

RFC 6265 was written by Adam Barth. This document is an update of RFC 6265, adding features and aligning the specification with the reality of today's deployments. Here, we're standing upon the shoulders of a giant since the majority of the text is still Adam's.

## Authors' Addresses

Lily Chen (editor)  
Google LLC  
Email: chlily@google.com

Steven Englehardt (editor)  
Mozilla  
Email: senglehardt@mozilla.com

Mike West (editor)  
Google LLC  
Email: mkwst@google.com  
URI: <https://mikewest.org/>

John Wilander (editor)  
Apple, Inc  
Email: wilander@apple.com



QUIC  
Internet-Draft  
Intended status: Standards Track  
Expires: 6 August 2021

M. Bishop, Ed.  
Akamai  
2 February 2021

Hypertext Transfer Protocol Version 3 (HTTP/3)  
draft-ietf-quic-http-34

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to HTTP/3.

DO NOT DEPLOY THIS VERSION OF HTTP

DO NOT DEPLOY THIS VERSION OF HTTP/3 UNTIL IT IS IN AN RFC. This version is still a work in progress. For trial deployments, please use earlier versions.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list ([quic@ietf.org](mailto:quic@ietf.org)), which is archived at [https://mailarchive.ietf.org/arch/search/?email\\_list=quic](https://mailarchive.ietf.org/arch/search/?email_list=quic).

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-http>.

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 <https://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 6 August 2021.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	4
1.1. Prior versions of HTTP . . . . .	5
1.2. Delegation to QUIC . . . . .	5
2. HTTP/3 Protocol Overview . . . . .	5
2.1. Document Organization . . . . .	6
2.2. Conventions and Terminology . . . . .	7
3. Connection Setup and Management . . . . .	8
3.1. Discovering an HTTP/3 Endpoint . . . . .	8
3.1.1. HTTP Alternative Services . . . . .	9
3.1.2. Other Schemes . . . . .	10
3.2. Connection Establishment . . . . .	10
3.3. Connection Reuse . . . . .	11
4. HTTP Request Lifecycle . . . . .	12
4.1. HTTP Message Exchanges . . . . .	12
4.1.1. Field Formatting and Compression . . . . .	14
4.1.2. Request Cancellation and Rejection . . . . .	17
4.1.3. Malformed Requests and Responses . . . . .	18
4.2. The CONNECT Method . . . . .	19
4.3. HTTP Upgrade . . . . .	21
4.4. Server Push . . . . .	21
5. Connection Closure . . . . .	23
5.1. Idle Connections . . . . .	23
5.2. Connection Shutdown . . . . .	24
5.3. Immediate Application Closure . . . . .	26
5.4. Transport Closure . . . . .	26
6. Stream Mapping and Usage . . . . .	27
6.1. Bidirectional Streams . . . . .	27
6.2. Unidirectional Streams . . . . .	28
6.2.1. Control Streams . . . . .	29
6.2.2. Push Streams . . . . .	30

6.2.3. Reserved Stream Types . . . . .	30
7. HTTP Framing Layer . . . . .	31
7.1. Frame Layout . . . . .	32
7.2. Frame Definitions . . . . .	32
7.2.1. DATA . . . . .	32
7.2.2. HEADERS . . . . .	33
7.2.3. CANCEL_PUSH . . . . .	33
7.2.4. SETTINGS . . . . .	35
7.2.5. PUSH_PROMISE . . . . .	38
7.2.6. GOAWAY . . . . .	39
7.2.7. MAX_PUSH_ID . . . . .	40
7.2.8. Reserved Frame Types . . . . .	41
8. Error Handling . . . . .	41
8.1. HTTP/3 Error Codes . . . . .	42
9. Extensions to HTTP/3 . . . . .	43
10. Security Considerations . . . . .	44
10.1. Server Authority . . . . .	44
10.2. Cross-Protocol Attacks . . . . .	44
10.3. Intermediary Encapsulation Attacks . . . . .	45
10.4. Cacheability of Pushed Responses . . . . .	45
10.5. Denial-of-Service Considerations . . . . .	45
10.5.1. Limits on Field Section Size . . . . .	46
10.5.2. CONNECT Issues . . . . .	47
10.6. Use of Compression . . . . .	47
10.7. Padding and Traffic Analysis . . . . .	48
10.8. Frame Parsing . . . . .	48
10.9. Early Data . . . . .	49
10.10. Migration . . . . .	49
10.11. Privacy Considerations . . . . .	49
11. IANA Considerations . . . . .	49
11.1. Registration of HTTP/3 Identification String . . . . .	50
11.2. New Registries . . . . .	50
11.2.1. Frame Types . . . . .	50
11.2.2. Settings Parameters . . . . .	52
11.2.3. Error Codes . . . . .	53
11.2.4. Stream Types . . . . .	55
12. References . . . . .	56
12.1. Normative References . . . . .	56
12.2. Informative References . . . . .	57
Appendix A. Considerations for Transitioning from HTTP/2 . . . . .	58
A.1. Streams . . . . .	59
A.2. HTTP Frame Types . . . . .	60
A.2.1. Prioritization Differences . . . . .	60
A.2.2. Field Compression Differences . . . . .	60
A.2.3. Flow Control Differences . . . . .	61
A.2.4. Guidance for New Frame Type Definitions . . . . .	61
A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types . . . . .	61
A.3. HTTP/2 SETTINGS Parameters . . . . .	62

A.4. HTTP/2 Error Codes . . . . .	64
A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors . . . . .	65
Appendix B. Change Log . . . . .	65
B.1. Since draft-ietf-quic-http-32 . . . . .	65
B.2. Since draft-ietf-quic-http-31 . . . . .	66
B.3. Since draft-ietf-quic-http-30 . . . . .	66
B.4. Since draft-ietf-quic-http-29 . . . . .	66
B.5. Since draft-ietf-quic-http-28 . . . . .	66
B.6. Since draft-ietf-quic-http-27 . . . . .	66
B.7. Since draft-ietf-quic-http-26 . . . . .	66
B.8. Since draft-ietf-quic-http-25 . . . . .	66
B.9. Since draft-ietf-quic-http-24 . . . . .	67
B.10. Since draft-ietf-quic-http-23 . . . . .	67
B.11. Since draft-ietf-quic-http-22 . . . . .	67
B.12. Since draft-ietf-quic-http-21 . . . . .	68
B.13. Since draft-ietf-quic-http-20 . . . . .	68
B.14. Since draft-ietf-quic-http-19 . . . . .	69
B.15. Since draft-ietf-quic-http-18 . . . . .	69
B.16. Since draft-ietf-quic-http-17 . . . . .	69
B.17. Since draft-ietf-quic-http-16 . . . . .	70
B.18. Since draft-ietf-quic-http-15 . . . . .	70
B.19. Since draft-ietf-quic-http-14 . . . . .	70
B.20. Since draft-ietf-quic-http-13 . . . . .	70
B.21. Since draft-ietf-quic-http-12 . . . . .	71
B.22. Since draft-ietf-quic-http-11 . . . . .	71
B.23. Since draft-ietf-quic-http-10 . . . . .	71
B.24. Since draft-ietf-quic-http-09 . . . . .	71
B.25. Since draft-ietf-quic-http-08 . . . . .	72
B.26. Since draft-ietf-quic-http-07 . . . . .	72
B.27. Since draft-ietf-quic-http-06 . . . . .	72
B.28. Since draft-ietf-quic-http-05 . . . . .	72
B.29. Since draft-ietf-quic-http-04 . . . . .	72
B.30. Since draft-ietf-quic-http-03 . . . . .	72
B.31. Since draft-ietf-quic-http-02 . . . . .	73
B.32. Since draft-ietf-quic-http-01 . . . . .	73
B.33. Since draft-ietf-quic-http-00 . . . . .	73
B.34. Since draft-shade-quic-http2-mapping-00 . . . . .	74
Acknowledgments . . . . .	74
Author's Address . . . . .	75

## 1. Introduction

HTTP semantics ([SEMANTICS]) are used for a broad range of services on the Internet. These semantics have most commonly been used with HTTP/1.1 and HTTP/2. HTTP/1.1 has been used over a variety of transport and session layers, while HTTP/2 has been used primarily with TLS over TCP. HTTP/3 supports the same semantics over a new transport protocol, QUIC.

### 1.1. Prior versions of HTTP

HTTP/1.1 ([HTTP11]) uses whitespace-delimited text fields to convey HTTP messages. While these exchanges are human-readable, using whitespace for message formatting leads to parsing complexity and excessive tolerance of variant behavior.

Because HTTP/1.1 does not include a multiplexing layer, multiple TCP connections are often used to service requests in parallel. However, that has a negative impact on congestion control and network efficiency, since TCP does not share congestion control across multiple connections.

HTTP/2 ([HTTP2]) introduced a binary framing and multiplexing layer to improve latency without modifying the transport layer. However, because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was directly impacted by the lost packet.

### 1.2. Delegation to QUIC

The QUIC transport protocol incorporates stream multiplexing and per-stream flow control, similar to that provided by the HTTP/2 framing layer. By providing reliability at the stream level and congestion control across the entire connection, QUIC has the capability to improve the performance of HTTP compared to a TCP mapping. QUIC also incorporates TLS 1.3 ([TLS13]) at the transport layer, offering comparable confidentiality and integrity to running TLS over TCP, with the improved connection setup latency of TCP Fast Open ([TFO]).

This document defines HTTP/3, a mapping of HTTP semantics over the QUIC transport protocol, drawing heavily on the design of HTTP/2. HTTP/3 relies on QUIC to provide confidentiality and integrity protection of data; peer authentication; and reliable, in-order, per-stream delivery. While delegating stream lifetime and flow control issues to QUIC, a binary framing similar to the HTTP/2 framing is used on each stream. Some HTTP/2 features are subsumed by QUIC, while other features are implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [HTTP2].

## 2. HTTP/3 Protocol Overview

HTTP/3 provides a transport for HTTP semantics using the QUIC transport protocol and an internal framing layer similar to HTTP/2.

Once a client knows that an HTTP/3 server exists at a certain endpoint, it opens a QUIC connection. QUIC provides protocol negotiation, stream-based multiplexing, and flow control. Discovery of an HTTP/3 endpoint is described in Section 3.1.

Within each stream, the basic unit of HTTP/3 communication is a frame (Section 7.2). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 4.1). Frames that apply to the entire connection are conveyed on a dedicated control stream.

Multiplexing of requests is performed using the QUIC stream abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

Server push is an interaction mode introduced in HTTP/2 ([HTTP2]) that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. Several HTTP/3 frames are used to manage server push, such as PUSH\_PROMISE, MAX\_PUSH\_ID, and CANCEL\_PUSH.

As in HTTP/2, request and response fields are compressed for transmission. Because HPACK ([HPACK]) relies on in-order transmission of compressed field sections (a guarantee not provided by QUIC), HTTP/3 replaces HPACK with QPACK ([QPACK]). QPACK uses separate unidirectional streams to modify and track field table state, while encoded field sections refer to the state of the table without modifying it.

## 2.1. Document Organization

The following sections provide a detailed overview of the lifecycle of an HTTP/3 connection:

- \* Connection Setup and Management (Section 3) covers how an HTTP/3 endpoint is discovered and an HTTP/3 connection is established.
- \* HTTP Request Lifecycle (Section 4) describes how HTTP semantics are expressed using frames.
- \* Connection Closure (Section 5) describes how HTTP/3 connections are terminated, either gracefully or abruptly.

The details of the wire protocol and interactions with the transport are described in subsequent sections:

- \* Stream Mapping and Usage (Section 6) describes the way QUIC streams are used.
- \* HTTP Framing Layer (Section 7) describes the frames used on most streams.
- \* Error Handling (Section 8) describes how error conditions are handled and expressed, either on a particular stream or for the connection as a whole.

Additional resources are provided in the final sections:

- \* Extensions to HTTP/3 (Section 9) describes how new capabilities can be added in future documents.
- \* A more detailed comparison between HTTP/2 and HTTP/3 can be found in Appendix A.

## 2.2. Conventions and Terminology

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.

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].

The following terms are used:

**abort:** An abrupt termination of a connection or stream, possibly due to an error condition.

**client:** The endpoint that initiates an HTTP/3 connection. Clients send HTTP requests and receive HTTP responses.

**connection:** A transport-layer connection between two endpoints, using QUIC as the transport protocol.

**connection error:** An error that affects the entire HTTP/3 connection.

**endpoint:** Either the client or server of the connection.

**frame:** The smallest unit of communication on a stream in HTTP/3, consisting of a header and a variable-length sequence of bytes structured according to the frame type.

Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC CONNECTION\_CLOSE frames." References without this preface refer to frames defined in Section 7.2.

**HTTP/3 connection:** A QUIC connection where the negotiated application protocol is HTTP/3.

**peer:** An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

**receiver:** An endpoint that is receiving frames.

**sender:** An endpoint that is transmitting frames.

**server:** The endpoint that accepts an HTTP/3 connection. Servers receive HTTP requests and send HTTP responses.

**stream:** A bidirectional or unidirectional bytestream provided by the QUIC transport. All streams within an HTTP/3 connection can be considered "HTTP/3 streams," but multiple stream types are defined within HTTP/3.

**stream error:** An application-level error on the individual stream.

The term "content" is defined in Section 6.4 of [SEMANTICS].

Finally, the terms "resource", "message", "user agent", "origin server", "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 3 of [SEMANTICS].

Packet diagrams in this document use the format defined in Section 1.3 of [QUIC-TRANSPORT] to illustrate the order and size of fields.

### 3. Connection Setup and Management

#### 3.1. Discovering an HTTP/3 Endpoint

HTTP relies on the notion of an authoritative response: a response that has been determined to be the most appropriate response for that request given the state of the target resource at the time of response message origination by (or at the direction of) the origin server identified within the target URI. Locating an authoritative server for an HTTP URI is discussed in Section 4.3 of [SEMANTICS].



The "https" scheme associates authority with possession of a certificate that the client considers to be trustworthy for the host identified by the authority component of the URI. Upon receiving a server certificate in the TLS handshake, the client **MUST** verify that the certificate is an acceptable match for the URI's origin server using the process described in Section 4.3.4 of [SEMANTICS]. If the certificate cannot be verified with respect to the URI's origin server, the client **MUST NOT** consider the server authoritative for that origin.

A client **MAY** attempt access to a resource with an "https" URI by resolving the host identifier to an IP address, establishing a QUIC connection to that address on the indicated port (including validation of the server certificate as described above), and sending an HTTP/3 request message targeting the URI to the server over that secured connection. Unless some other mechanism is used to select HTTP/3, the token "h3" is used in the Application Layer Protocol Negotiation (ALPN; see [RFC7301]) extension during the TLS handshake.

Connectivity problems (e.g., blocking UDP) can result in QUIC connection establishment failure; clients **SHOULD** attempt to use TCP-based versions of HTTP in this case.

Servers **MAY** serve HTTP/3 on any UDP port; an alternative service advertisement always includes an explicit port, and URIs contain either an explicit port or a default port associated with the scheme.

#### 3.1.1. HTTP Alternative Services

An HTTP origin can advertise the availability of an equivalent HTTP/3 endpoint via the Alt-Svc HTTP response header field or the HTTP/2 ALTSVC frame ([ALTSVC]), using the "h3" ALPN token.

For example, an origin could indicate in an HTTP response that HTTP/3 was available on UDP port 50781 at the same hostname by including the following header field:

```
Alt-Svc: h3=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/3 support, a client **MAY** attempt to establish a QUIC connection to the indicated host and port; if this connection is successful, the client can send HTTP requests using the mapping described in this document.

### 3.1.2. Other Schemes

Although HTTP is independent of the transport protocol, the "http" scheme associates authority with the ability to receive TCP connections on the indicated port of whatever host is identified within the authority component. Because HTTP/3 does not use TCP, HTTP/3 cannot be used for direct access to the authoritative server for a resource identified by an "http" URI. However, protocol extensions such as [ALTSVC] permit the authoritative server to identify other services that are also authoritative and that might be reachable over HTTP/3.

Prior to making requests for an origin whose scheme is not "https", the client **MUST** ensure the server is willing to serve that scheme. For origins whose scheme is "http", an experimental method to accomplish this is described in [RFC8164]. Other mechanisms might be defined for various schemes in the future.

### 3.2. Connection Establishment

HTTP/3 relies on QUIC version 1 as the underlying transport. The use of other QUIC transport versions with HTTP/3 **MAY** be defined by future specifications.

QUIC version 1 uses TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients **MUST** support a mechanism to indicate the target host to the server during the TLS handshake. If the server is identified by a domain name ([DNS-TERMS]), clients **MUST** send the Server Name Indication (SNI; [RFC6066]) TLS extension unless an alternative mechanism to indicate the target host is used.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/3 support is indicated by selecting the ALPN token "h3" in the TLS handshake. Support for other application-layer protocols **MAY** be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP/3-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 7.2.4) **MUST** be sent by each endpoint as the initial frame of their respective HTTP control stream; see Section 6.2.1.

### 3.3. Connection Reuse

HTTP/3 connections are persistent across multiple requests. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. To use an existing connection for a new origin, clients MUST validate the certificate presented by the server for the new origin server using the process described in Section 4.3.4 of [SEMANTICS]. This implies that clients will need to retain the server certificate and any additional information needed to verify that certificate; clients which do not do so will be unable to reuse the connection for additional origins.

If the certificate is not acceptable with regard to the new origin for any reason, the connection MUST NOT be reused and a new connection SHOULD be established for the new origin. If the reason the certificate cannot be verified might apply to other origins already associated with the connection, the client SHOULD re-validate the server certificate for those origins. For instance, if validation of a certificate fails because the certificate has expired or been revoked, this might be used to invalidate all other origins for which that certificate was used to establish authority.

Clients SHOULD NOT open more than one HTTP/3 connection to a given IP address and UDP port, where the IP address and port might be derived from a URI, a selected alternative service ([ALTSVC]), a configured proxy, or name resolution of any of these. A client MAY open multiple HTTP/3 connections to the same IP address and UDP port using different transport or TLS configurations but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open HTTP/3 connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the HTTP/3 connection, the terminating endpoint SHOULD first send a GOAWAY frame (Section 5.2) so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

A server that does not wish clients to reuse HTTP/3 connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request; see Section 7.4 of [SEMANTICS].

## 4. HTTP Request Lifecycle

### 4.1. HTTP Message Exchanges

A client sends an HTTP request on a request stream, which is a client-initiated bidirectional QUIC stream; see Section 6.1. A client **MUST** send only a single request on a given stream. A server sends zero or more interim HTTP responses on the same stream as the request, followed by a single final HTTP response, as detailed below. See Section 15 of [SEMANTICS] for a description of interim and final HTTP responses.

Pushed responses are sent on a server-initiated unidirectional QUIC stream; see Section 6.2.2. A server sends zero or more interim HTTP responses, followed by a single final HTTP response, in the same manner as a standard response. Push is described in more detail in Section 4.4.

On a given stream, receipt of multiple requests or receipt of an additional HTTP response following a final HTTP response **MUST** be treated as malformed (Section 4.1.3).

An HTTP message (request or response) consists of:

1. the header section, sent as a single HEADERS frame (see Section 7.2.2),
2. optionally, the content, if present, sent as a series of DATA frames (see Section 7.2.1), and
3. optionally, the trailer section, if present, sent as a single HEADERS frame.

Header and trailer sections are described in Sections 6.3 and 6.5 of [SEMANTICS]; the content is described in Section 6.4 of [SEMANTICS].

Receipt of an invalid sequence of frames **MUST** be treated as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8. In particular, a DATA frame before any HEADERS frame, or a HEADERS or DATA frame after the trailing HEADERS frame, is considered invalid. Other frame types, especially unknown frame types, might be permitted subject to their own rules; see Section 9.

A server MAY send one or more PUSH\_PROMISE frames (Section 7.2.5) before, after, or interleaved with the frames of a response message. These PUSH\_PROMISE frames are not part of the response; see Section 4.4 for more details. PUSH\_PROMISE frames are not permitted on push streams; a pushed response that includes PUSH\_PROMISE frames MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

Frames of unknown types (Section 9), including reserved frames (Section 7.2.8) MAY be sent on a request or push stream before, after, or interleaved with other frames described in this section.

The HEADERS and PUSH\_PROMISE frames might reference updates to the QPACK dynamic table. While these updates are not directly part of the message exchange, they must be received and processed before the message can be consumed. See Section 4.1.1 for more details.

Transfer codings (see Section 6.1 of [HTTP11]) are not defined for HTTP/3; the Transfer-Encoding header field MUST NOT be used.

A response MAY consist of multiple messages when and only when one or more interim responses (1xx; see Section 15.2 of [SEMANTICS]) precede a final response to the same request. Interim responses do not contain content or trailer sections.

An HTTP request/response exchange fully consumes a client-initiated bidirectional QUIC stream. After sending a request, a client MUST close the stream for sending. Unless using the CONNECT method (see Section 4.2), clients MUST NOT make stream closure dependent on receiving a response to their request. After sending a final response, the server MUST close the stream for sending. At this point, the QUIC stream is fully closed.

When a stream is closed, this indicates the end of the final HTTP message. Because some messages are large or unbounded, endpoints SHOULD begin processing partial HTTP messages once enough of the message has been received to make progress. If a client-initiated stream terminates without enough of the HTTP message to provide a complete response, the server SHOULD abort its response stream with the error code H3\_REQUEST\_INCOMPLETE; see Section 8.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When the server does not need to receive the remainder of the request, it MAY abort reading the request stream, send a complete response, and cleanly close the sending part of the stream. The error code H3\_NO\_ERROR SHOULD be used when requesting that the client stop sending on the

request stream. Clients MUST NOT discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. If the server sends a partial or complete response but does not abort reading the request, clients SHOULD continue sending the body of the request and close the stream normally.

#### 4.1.1. Field Formatting and Compression

HTTP messages carry metadata as a series of key-value pairs called HTTP fields; see Sections 6.3 and 6.5 of [SEMANTICS]. For a listing of registered HTTP fields, see the "Hypertext Transfer Protocol (HTTP) Field Name Registry" maintained at <https://www.iana.org/assignments/http-fields/>.

*\*Note:* This registry will not exist until [SEMANTICS] is approved. *\*RFC Editor\**, please remove this note prior to publication.

Field names are strings containing a subset of ASCII characters. Properties of HTTP field names and values are discussed in more detail in Section 5.1 of [SEMANTICS]. As in HTTP/2, characters in field names MUST be converted to lowercase prior to their encoding. A request or response containing uppercase characters in field names MUST be treated as malformed (Section 4.1.3).

Like HTTP/2, HTTP/3 does not use the Connection header field to indicate connection-specific fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/3 field section containing connection-specific fields; any message containing connection-specific fields MUST be treated as malformed (Section 4.1.3).

The only exception to this is the TE header field, which MAY be present in an HTTP/3 request header; when it is, it MUST NOT contain any value other than "trailers".

An intermediary transforming an HTTP/1.x message to HTTP/3 MUST remove connection-specific header fields as discussed in Section 7.6.1 of [SEMANTICS], or their messages will be treated by other HTTP/3 endpoints as malformed (Section 4.1.3).

##### 4.1.1.1. Pseudo-Header Fields

Like HTTP/2, HTTP/3 employs a series of pseudo-header fields where the field name begins with the ':' character (ASCII 0x3a). These pseudo-header fields convey the target URI, the method of the request, and the status code for the response.

Pseudo-header fields are not HTTP fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document; however, an extension could negotiate a modification of this restriction; see Section 9.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailer sections. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 4.1.3).

All pseudo-header fields MUST appear in the header section before regular header fields. Any request or response that contains a pseudo-header field that appears in a header section after a regular header field MUST be treated as malformed (Section 4.1.3).

The following pseudo-header fields are defined for requests:

**":method":** Contains the HTTP method (Section 9 of [SEMANTICS])

**":scheme":** Contains the scheme portion of the target URI (Section 3.1 of [URI])

**":scheme"** is not restricted to URIs with scheme "http" and "https". A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

See Section 3.1.2 for guidance on using a scheme other than "https".

**":authority":** Contains the authority portion of the target URI (Section 3.2 of [URI]). The authority MUST NOT include the deprecated "userinfo" subcomponent for URIs of scheme "http" or "https".

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form; see Section 7.1 of [SEMANTICS]. Clients that generate HTTP/3 requests directly SHOULD use the **":authority"** pseudo-header field instead of the Host field. An intermediary that converts an HTTP/3 request to HTTP/1.1 MUST create a Host field if one is not present in a request by copying the value of the **":authority"** pseudo-header field.

`":path"`: Contains the path and query parts of the target URI (the `"path-absolute"` production and optionally a `'?'` character followed by the `"query"` production; see Sections 3.3 and 3.4 of [URI]. A request in asterisk form includes the value `'*'` for the `":path"` pseudo-header field.

This pseudo-header field MUST NOT be empty for `"http"` or `"https"` URIs; `"http"` or `"https"` URIs that do not contain a path component MUST include a value of `'/'`. The exception to this rule is an `OPTIONS` request for an `"http"` or `"https"` URI that does not include a path component; these MUST include a `":path"` pseudo-header field with a value of `'*'`; see Section 7.1 of [SEMANTICS].

All HTTP/3 requests MUST include exactly one value for the `":method"`, `":scheme"`, and `":path"` pseudo-header fields, unless it is a `CONNECT` request; see Section 4.2.

If the `":scheme"` pseudo-header field identifies a scheme that has a mandatory authority component (including `"http"` and `"https"`), the request MUST contain either an `":authority"` pseudo-header field or a `"Host"` header field. If these fields are present, they MUST NOT be empty. If both fields are present, they MUST contain the same value. If the scheme does not have a mandatory authority component and none is provided in the request target, the request MUST NOT contain the `":authority"` pseudo-header or `"Host"` header fields.

An HTTP request that omits mandatory pseudo-header fields or contains invalid values for those pseudo-header fields is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

For responses, a single `":status"` pseudo-header field is defined that carries the HTTP status code; see Section 15 of [SEMANTICS]. This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.



#### 4.1.1.2. Field Compression

[QPACK] describes a variation of HPACK that gives an encoder some control over how much head-of-line blocking can be caused by compression. This allows an encoder to balance compression efficiency with latency. HTTP/3 uses QPACK to compress header and trailer sections, including the pseudo-header fields present in the header section.

To allow for better compression efficiency, the "Cookie" field ([RFC6265]) MAY be split into separate field lines, each with one or more cookie-pairs, before compression. If a decompressed field section contains multiple cookie field lines, these MUST be concatenated into a single byte string using the two-byte delimiter of 0x3b, 0x20 (the ASCII string "; ") before being passed into a context other than HTTP/2 or HTTP/3, such as an HTTP/1.1 connection, or a generic HTTP server application.

#### 4.1.1.3. Header Size Constraints

An HTTP/3 implementation MAY impose a limit on the maximum size of the message header it will accept on an individual HTTP message. A server that receives a larger header section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process. The size of a field list is calculated based on the uncompressed size of fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each field.

If an implementation wishes to advise its peer of this limit, it can be conveyed as a number of bytes in the SETTINGS\_MAX\_FIELD\_SECTION\_SIZE parameter. An implementation that has received this parameter SHOULD NOT send an HTTP message header that exceeds the indicated size, as the peer will likely refuse to process it. However, an HTTP message can traverse one or more intermediaries before reaching the origin server; see Section 3.7 of [SEMANTICS]. Because this limit is applied separately by each implementation which processes the message, messages below this limit are not guaranteed to be accepted.

#### 4.1.2. Request Cancellation and Rejection

Once a request stream has been opened, the request MAY be cancelled by either endpoint. Clients cancel requests if the response is no longer of interest; servers cancel requests if they are unable to or choose not to respond. When possible, it is RECOMMENDED that servers send an HTTP response with an appropriate status code rather than canceling a request it has already begun processing.

Implementations SHOULD cancel requests by abruptly terminating any directions of a stream that are still open. This means resetting the sending parts of streams and aborting reading on receiving parts of streams; see Section 2.4 of [QUIC-TRANSPORT].

When the server cancels a request without performing any application processing, the request is considered "rejected." The server SHOULD abort its response stream with the error code `H3_REQUEST_REJECTED`. In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result. The client can treat requests rejected by the server as though they had never been sent at all, thereby allowing them to be retried later.

Servers MUST NOT use the `H3_REQUEST_REJECTED` error code for requests that were partially or fully processed. When a server abandons a response after partial processing, it SHOULD abort its response stream with the error code `H3_REQUEST_CANCELLED`.

Client SHOULD use the error code `H3_REQUEST_CANCELLED` to cancel requests. Upon receipt of this error code, a server MAY abruptly terminate the response using the error code `H3_REQUEST_REJECTED` if no processing was performed. Clients MUST NOT use the `H3_REQUEST_REJECTED` error code, except when a server has requested closure of the request stream with this error code.

If a stream is canceled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Only idempotent actions such as GET, PUT, or DELETE can be safely retried; a client SHOULD NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are idempotent independent of the method or some means to detect that the original request was never applied. See Section 9.2.2 of [SEMANTICS] for more details.

#### 4.1.3. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of frames but is invalid due to:

- \* the presence of prohibited fields or pseudo-header fields,
- \* the absence of mandatory pseudo-header fields,
- \* invalid values for pseudo-header fields,

- \* pseudo-header fields after fields,
- \* an invalid sequence of HTTP messages,
- \* the inclusion of uppercase field names, or
- \* the inclusion of invalid characters in field names or values.

A request or response that is defined as having content when it contains a Content-Length header field (Section 6.4.1 of [SEMANTICS]), is malformed if the value of a Content-Length header field does not equal the sum of the DATA frame lengths received. A response that is defined as never having content, even when a Content-Length is present, can have a non-zero Content-Length field even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 8) of type H3\_MESSAGE\_ERROR.

For malformed requests, a server MAY send an HTTP response indicating the error prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

#### 4.2. The CONNECT Method

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target; see Section 9.3.6 of [SEMANTICS]. It is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2 and HTTP/3, the CONNECT method is used to establish a tunnel over a single stream.

A CONNECT request MUST be constructed as follows:

- \* The ":method" pseudo-header field is set to "CONNECT"
- \* The ":scheme" and ":path" pseudo-header fields are omitted

- \* The ":authority" pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests; see Section 7.1 of [SEMANTICS])

The request stream remains open at the end of the request to carry the data to be transferred. A CONNECT request that does not conform to these restrictions is malformed; see Section 4.1.3.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in Section 15.3 of [SEMANTICS].

All DATA frames on the stream correspond to data sent or received on the TCP connection. The payload of any DATA frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

Once the CONNECT method has completed, only DATA frames are permitted to be sent on the stream. Extension frames MAY be used if specifically permitted by the definition of the extension. Receipt of any other known frame type MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will close the send stream that it sends to the client. TCP connections that remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT close a stream for sending while they still expect to receive data from the target of the CONNECT.

A TCP connection error is signaled by abruptly terminating the stream. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type H3\_CONNECT\_ERROR; see Section 8. Correspondingly, if a proxy detects an error with the stream or the QUIC connection, it MUST close the TCP connection. If the underlying TCP implementation permits it, the proxy SHOULD send a TCP segment with the RST bit set.

Since CONNECT creates a tunnel to an arbitrary server, proxies that support CONNECT SHOULD restrict its use to a set of known ports or a list of safe request targets; see Section 9.3.6 of [SEMANTICS] for more detail.

#### 4.3. HTTP Upgrade

HTTP/3 does not support the HTTP Upgrade mechanism (Section 7.8 of [SEMANTICS]) or 101 (Switching Protocols) informational status code (Section 15.2.2 of [SEMANTICS]).

#### 4.4. Server Push

Server push is an interaction mode that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. HTTP/3 server push is similar to what is described in Section 8.2 of [HTTP2], but uses different mechanisms.

Each server push is assigned a unique Push ID by the server. The Push ID is used to refer to the push in various contexts throughout the lifetime of the HTTP/3 connection.

The Push ID space begins at zero, and ends at a maximum value set by the MAX\_PUSH\_ID frame; see Section 7.2.7. In particular, a server is not able to push until after the client sends a MAX\_PUSH\_ID frame. A client sends MAX\_PUSH\_ID frames to control the number of pushes that a server can promise. A server SHOULD use Push IDs sequentially, beginning from zero. A client MUST treat receipt of a push stream as a connection error of type H3\_ID\_ERROR (Section 8) when no MAX\_PUSH\_ID frame has been sent or when the stream references a Push ID that is greater than the maximum Push ID.

The Push ID is used in one or more PUSH\_PROMISE frames (Section 7.2.5) that carry the header section of the request message. These frames are sent on the request stream that generated the push. This allows the server push to be associated with a client request. When the same Push ID is promised on multiple request streams, the decompressed request field sections MUST contain the same fields in the same order, and both the name and the value in each field MUST be identical.

The Push ID is then included with the push stream that ultimately fulfills those promises; see Section 6.2.2. The push stream identifies the Push ID of the promise that it fulfills, then contains a response to the promised request as described in Section 4.1.

Finally, the Push ID can be used in CANCEL\_PUSH frames; see Section 7.2.3. Clients use this frame to indicate they do not wish to receive a promised resource. Servers use this frame to indicate they will not be fulfilling a previous promise.

Not all requests can be pushed. A server MAY push requests that have the following properties:

- \* cacheable; see Section 9.2.3 of [SEMANTICS]
- \* safe; see Section 9.2.1 of [SEMANTICS]
- \* does not include a request body or trailer section

The server MUST include a value in the ":authority" pseudo-header field for which the server is authoritative. If the client has not yet validated the connection for the origin indicated by the pushed request, it MUST perform the same verification process it would do before sending a request for that origin on the connection; see Section 3.3. If this verification fails, the client MUST NOT consider the server authoritative for that origin.

Clients SHOULD send a CANCEL\_PUSH frame upon receipt of a PUSH\_PROMISE frame carrying a request that is not cacheable, is not known to be safe, that indicates the presence of a request body, or for which it does not consider the server authoritative. Any corresponding responses MUST NOT be used or cached.

Each pushed response is associated with one or more client requests. The push is associated with the request stream on which the PUSH\_PROMISE frame was received. The same server push can be associated with additional client requests using a PUSH\_PROMISE frame with the same Push ID on multiple request streams. These associations do not affect the operation of the protocol, but MAY be considered by user agents when deciding how to use pushed resources.

Ordering of a PUSH\_PROMISE frame in relation to certain parts of the response is important. The server SHOULD send PUSH\_PROMISE frames prior to sending HEADERS or DATA frames that reference the promised responses. This reduces the chance that a client requests a resource that will be pushed by the server.

Due to reordering, push stream data can arrive before the corresponding PUSH\_PROMISE frame. When a client receives a new push stream with an as-yet-unknown Push ID, both the associated client request and the pushed request header fields are unknown. The client can buffer the stream data in expectation of the matching PUSH\_PROMISE. The client can use stream flow control (see Section 4.1 of [QUIC-TRANSPORT]) to limit the amount of data a server may commit to the pushed stream.

Push stream data can also arrive after a client has canceled a push. In this case, the client can abort reading the stream with an error code of H3\_REQUEST\_CANCELLED. This asks the server not to transfer additional data and indicates that it will be discarded upon receipt.

Pushed responses that are cacheable (see Section 3 of [CACHING]) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present; see Section 5.2.2.3 of [CACHING]) at the time the pushed response is received.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

## 5. Connection Closure

Once established, an HTTP/3 connection can be used for many requests and responses over time until the connection is closed. Connection closure can happen in any of several different ways.

### 5.1. Idle Connections

Each QUIC endpoint declares an idle timeout during the handshake. If the QUIC connection remains idle (no packets received) for longer than this duration, the peer will assume that the connection has been closed. HTTP/3 implementations will need to open a new HTTP/3 connection for new requests if the existing connection has been idle for longer than the idle timeout negotiated during the QUIC handshake, and SHOULD do so if approaching the idle timeout; see Section 10.1 of [QUIC-TRANSPORT].

HTTP clients are expected to request that the transport keep connections open while there are responses outstanding for requests or server pushes, as described in Section 10.1.2 of [QUIC-TRANSPORT]. If the client is not expecting a response from the server, allowing an idle connection to time out is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY

maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers. Servers SHOULD NOT actively keep connections open.

## 5.2. Connection Shutdown

Even when a connection is not idle, either endpoint can decide to stop using the connection and initiate a graceful connection close. Endpoints initiate the graceful shutdown of an HTTP/3 connection by sending a GOAWAY frame (Section 7.2.6). The GOAWAY frame contains an identifier that indicates to the receiver the range of requests or pushes that were or might be processed in this connection. The server sends a client-initiated bidirectional Stream ID; the client sends a Push ID (Section 4.4). Requests or pushes with the indicated identifier or greater are rejected (Section 4.1.2) by the sender of the GOAWAY. This identifier MAY be zero if no requests or pushes were processed.

The information in the GOAWAY frame enables a client and server to agree on which requests or pushes were accepted prior to the shutdown of the HTTP/3 connection. Upon sending a GOAWAY frame, the endpoint SHOULD explicitly cancel (see Section 4.1.2 and Section 7.2.3) any requests or pushes that have identifiers greater than or equal to that indicated, in order to clean up transport state for the affected streams. The endpoint SHOULD continue to do so as more requests or pushes arrive.

Endpoints MUST NOT initiate new requests or promise new pushes on the connection after receipt of a GOAWAY frame from the peer. Clients MAY establish a new connection to send additional requests.

Some requests or pushes might already be in transit:

- \* Upon receipt of a GOAWAY frame, if the client has already sent requests with a Stream ID greater than or equal to the identifier contained in the GOAWAY frame, those requests will not be processed. Clients can safely retry unprocessed requests on a different HTTP connection. A client that is unable to retry requests loses all requests that are in flight when the server closes the connection.

Requests on Stream IDs less than the Stream ID in a GOAWAY frame from the server might have been processed; their status cannot be known until a response is received, the stream is reset individually, another GOAWAY is received with a lower Stream ID than that of the request in question, or the connection terminates.



Servers MAY reject individual requests on streams below the indicated ID if these requests were not processed.

- \* If a server receives a GOAWAY frame after having promised pushes with a Push ID greater than or equal to the identifier contained in the GOAWAY frame, those pushes will not be accepted.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a request has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint MAY send multiple GOAWAY frames indicating different identifiers, but the identifier in each frame MUST NOT be greater than the identifier in any previous frame, since clients might already have retried unprocessed requests on another HTTP connection. Receiving a GOAWAY containing a larger identifier than previously received MUST be treated as a connection error of type H3\_ID\_ERROR; see Section 8.

An endpoint that is attempting to gracefully shut down a connection can send a GOAWAY frame with a value set to the maximum possible value ( $2^{62}-4$  for servers,  $2^{62}-1$  for clients). This ensures that the peer stops creating new requests or pushes. After allowing time for any in-flight requests or pushes to arrive, the endpoint can send another GOAWAY frame indicating which requests or pushes it might accept before the end of the connection. This ensures that a connection can be cleanly shut down without losing requests.

A client has more flexibility in the value it chooses for the Push ID in a GOAWAY that it sends. A value of  $2^{62}-1$  indicates that the server can continue fulfilling pushes that have already been promised. A smaller value indicates the client will reject pushes with Push IDs greater than or equal to this value. Like the server, the client MAY send subsequent GOAWAY frames so long as the specified Push ID is no greater than any previously sent value.

Even when a GOAWAY indicates that a given request or push will not be processed or accepted upon receipt, the underlying transport resources still exist. The endpoint that initiated these requests can cancel them to clean up transport state.

Once all accepted requests and pushes have been processed, the endpoint can permit the connection to become idle, or MAY initiate an immediate closure of the connection. An endpoint that completes a graceful shutdown SHOULD use the H3\_NO\_ERROR error code when closing the connection.

If a client has consumed all available bidirectional stream IDs with requests, the server need not send a GOAWAY frame, since the client is unable to make further requests.

### 5.3. Immediate Application Closure

An HTTP/3 implementation can immediately close the QUIC connection at any time. This results in sending a QUIC CONNECTION\_CLOSE frame to the peer indicating that the application layer has terminated the connection. The application error code in this frame indicates to the peer why the connection is being closed. See Section 8 for error codes that can be used when closing a connection in HTTP/3.

Before closing the connection, a GOAWAY frame MAY be sent to allow the client to retry some requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION\_CLOSE frame improves the chances of the frame being received by clients.

If there are open streams that have not been explicitly closed, they are implicitly closed when the connection is closed; see Section 10.2 of [QUIC-TRANSPORT].

### 5.4. Transport Closure

For various reasons, the QUIC transport could indicate to the application layer that the connection has terminated. This might be due to an explicit closure by the peer, a transport-level error, or a change in network topology that interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST assume that any request that was sent, whether in whole or in part, might have been processed.

## 6. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. In version 1 of QUIC, the stream data containing HTTP frames is carried by QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received stream data, exposing a reliable byte stream to the application. Although QUIC permits out-of-order delivery within a stream, HTTP/3 does not make use of this feature.

QUIC streams can be either unidirectional, carrying data only from initiator to receiver, or bidirectional. Streams can be initiated by either the client or the server. For more detail on QUIC streams, see Section 2 of [QUIC-TRANSPORT].

When HTTP fields and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction or to the entire HTTP/3 connection context.

### 6.1. Bidirectional Streams

All client-initiated bidirectional streams are used for HTTP requests and responses. A bidirectional stream ensures that the response can be readily correlated with the request. These streams are referred to as request streams.

This means that the client's first request occurs on QUIC stream 0, with subsequent requests on stream 4, 8, and so on. In order to permit these streams to open, an HTTP/3 server SHOULD configure non-zero minimum values for the number of permitted streams and the initial stream flow control window. So as to not unnecessarily limit parallelism, at least 100 request streams SHOULD be permitted at a time.

HTTP/3 does not use server-initiated bidirectional streams, though an extension could define a use for these streams. Clients MUST treat receipt of a server-initiated bidirectional stream as a connection error of type H3\_STREAM\_CREATION\_ERROR (Section 8) unless such an extension has been negotiated.

## 6.2. Unidirectional Streams

Unidirectional streams, in either direction, are used for a range of purposes. The purpose is indicated by a stream type, which is sent as a variable-length integer at the start of the stream. The format and structure of data that follows this integer is determined by the stream type.

```
Unidirectional Stream Header {  
    Stream Type (i),  
}
```

Figure 1: Unidirectional Stream Header

Two stream types are defined in this document: control streams (Section 6.2.1) and push streams (Section 6.2.2). [QPACK] defines two additional stream types. Other stream types can be defined by extensions to HTTP/3; see Section 9 for more details. Some stream types are reserved (Section 6.2.3).

The performance of HTTP/3 connections in the early phase of their lifetime is sensitive to the creation and exchange of data on unidirectional streams. Endpoints that excessively restrict the number of streams or the flow control window of these streams will increase the chance that the remote peer reaches the limit early and becomes blocked. In particular, implementations should consider that remote peers may wish to exercise reserved stream behavior (Section 6.2.3) with some of the unidirectional streams they are permitted to use. To avoid blocking, the transport parameters sent by both clients and servers **MUST** allow the peer to create at least one unidirectional stream for the HTTP control stream plus the number of unidirectional streams required by mandatory extensions (three being the minimum number required for the base HTTP/3 protocol and QPACK), and **SHOULD** provide at least 1,024 bytes of flow control credit to each stream.

Note that an endpoint is not required to grant additional credits to create more unidirectional streams if its peer consumes all the initial credits before creating the critical unidirectional streams. Endpoints **SHOULD** create the HTTP control stream as well as the unidirectional streams required by mandatory extensions (such as the QPACK encoder and decoder streams) first, and then create additional streams as allowed by their peer.

If the stream header indicates a stream type that is not supported by the recipient, the remainder of the stream cannot be consumed as the semantics are unknown. Recipients of unknown stream types MAY abort reading of the stream with an error code of `H3_STREAM_CREATION_ERROR` or a reserved error code (Section 8.1), but MUST NOT consider such streams to be a connection error of any kind.

Implementations MAY send stream types before knowing whether the peer supports them. However, stream types that could modify the state or semantics of existing protocol components, including QPACK or other extensions, MUST NOT be sent until the peer is known to support them.

A sender can close or reset a unidirectional stream unless otherwise specified. A receiver MUST tolerate unidirectional streams being closed or reset prior to the reception of the unidirectional stream header.

#### 6.2.1. Control Streams

A control stream is indicated by a stream type of `0x00`. Data on this stream consists of HTTP/3 frames, as defined in Section 7.2.

Each side MUST initiate a single control stream at the beginning of the connection and send its SETTINGS frame as the first frame on this stream. If the first frame of the control stream is any other frame type, this MUST be treated as a connection error of type `H3_MISSING_SETTINGS`. Only one control stream per peer is permitted; receipt of a second stream claiming to be a control stream MUST be treated as a connection error of type `H3_STREAM_CREATION_ERROR`. The sender MUST NOT close the control stream, and the receiver MUST NOT request that the sender close the control stream. If either control stream is closed at any point, this MUST be treated as a connection error of type `H3_CLOSED_CRITICAL_STREAM`. Connection errors are described in Section 8.

Because the contents of the control stream are used to manage the behavior of other streams, endpoints SHOULD provide enough flow control credit to keep the peer's control stream from becoming blocked.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon as it is able. Depending on whether 0-RTT is available on the QUIC connection, either client or server might be able to send stream data first.

### 6.2.2. Push Streams

Server push is an optional feature introduced in HTTP/2 that allows a server to initiate a response before a request has been made. See Section 4.4 for more details.

A push stream is indicated by a stream type of 0x01, followed by the Push ID of the promise that it fulfills, encoded as a variable-length integer. The remaining data on this stream consists of HTTP/3 frames, as defined in Section 7.2, and fulfills a promised server push by zero or more interim HTTP responses followed by a single final HTTP response, as defined in Section 4.1. Server push and Push IDs are described in Section 4.4.

Only servers can push; if a server receives a client-initiated push stream, this MUST be treated as a connection error of type H3\_STREAM\_CREATION\_ERROR; see Section 8.

```
Push Stream Header {  
    Stream Type (i) = 0x01,  
    Push ID (i),  
}
```

Figure 2: Push Stream Header

Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type H3\_ID\_ERROR; see Section 8.

### 6.2.3. Reserved Stream Types

Stream types of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored. These streams have no semantics, and can be sent when application-layer padding is desired. They MAY also be sent on connections where no data is currently being transferred. Endpoints MUST NOT consider these streams to have any meaning upon receipt.

The payload and length of the stream are selected in any manner the sending implementation chooses. When sending a reserved stream type, the implementation MAY either terminate the stream cleanly or reset it. When resetting the stream, either the H3\_NO\_ERROR error code or a reserved error code (Section 8.1) SHOULD be used.

## 7. HTTP Framing Layer

HTTP frames are carried on QUIC streams, as described in Section 6. HTTP/3 defines three stream types: control stream, request stream, and push stream. This section describes HTTP/3 frame formats and their permitted stream types; see Table 1 for an overview. A comparison between HTTP/2 and HTTP/3 frames is provided in Appendix A.2.

Frame	Control Stream	Request Stream	Push Stream	Section
DATA	No	Yes	Yes	Section 7.2.1
HEADERS	No	Yes	Yes	Section 7.2.2
CANCEL_PUSH	Yes	No	No	Section 7.2.3
SETTINGS	Yes (1)	No	No	Section 7.2.4
PUSH_PROMISE	No	Yes	No	Section 7.2.5
GOAWAY	Yes	No	No	Section 7.2.6
MAX_PUSH_ID	Yes	No	No	Section 7.2.7
Reserved	Yes	Yes	Yes	Section 7.2.8

Table 1: HTTP/3 Frames and Stream Type Overview

The SETTINGS frame can only occur as the first frame of a Control stream; this is indicated in Table 1 with a (1). Specific guidance is provided in the relevant section.

Note that, unlike QUIC frames, HTTP/3 frames can span multiple packets.

## 7.1. Frame Layout

All frames have the following format:

```
HTTP/3 Frame Format {  
    Type (i),  
    Length (i),  
    Frame Payload (...),  
}
```

Figure 3: HTTP/3 Frame Format

A frame includes the following fields:

**Type:** A variable-length integer that identifies the frame type.

**Length:** A variable-length integer that describes the length in bytes of the Frame Payload.

**Frame Payload:** A payload, the semantics of which are determined by the Type field.

Each frame's payload MUST contain exactly the fields identified in its description. A frame payload that contains additional bytes after the identified fields or a frame payload that terminates before the end of the identified fields MUST be treated as a connection error of type `H3_FRAME_ERROR`; see Section 8. In particular, redundant length encodings MUST be verified to be self-consistent; see Section 10.8.

When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error of type `H3_FRAME_ERROR`; see Section 8. Streams that terminate abruptly may be reset at any point in a frame.

## 7.2. Frame Definitions

### 7.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of bytes associated with HTTP request or response content.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on a control stream, the recipient MUST respond with a connection error of type `H3_FRAME_UNEXPECTED`; see Section 8.



```
DATA Frame {  
    Type (i) = 0x0,  
    Length (i),  
    Data (...),  
}
```

Figure 4: DATA Frame

#### 7.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry an HTTP field section, encoded using QPACK. See [QPACK] for more details.

```
HEADERS Frame {  
    Type (i) = 0x1,  
    Length (i),  
    Encoded Field Section (...),  
}
```

Figure 5: HEADERS Frame

HEADERS frames can only be sent on request or push streams. If a HEADERS frame is received on a control stream, the recipient **MUST** respond with a connection error (Section 8) of type `H3_FRAME_UNEXPECTED`.

#### 7.2.3. CANCEL\_PUSH

The CANCEL\_PUSH frame (type=0x3) is used to request cancellation of a server push prior to the push stream being received. The CANCEL\_PUSH frame identifies a server push by Push ID (see Section 4.4), encoded as a variable-length integer.

When a client sends CANCEL\_PUSH, it is indicating that it does not wish to receive the promised resource. The server **SHOULD** abort sending the resource, but the mechanism to do so depends on the state of the corresponding push stream. If the server has not yet created a push stream, it does not create one. If the push stream is open, the server **SHOULD** abruptly terminate that stream. If the push stream has already ended, the server **MAY** still abruptly terminate the stream or **MAY** take no action.

A server sends CANCEL\_PUSH to indicate that it will not be fulfilling a promise which was previously sent. The client cannot expect the corresponding promise to be fulfilled, unless it has already received and processed the promised response. Regardless of whether a push stream has been opened, a server SHOULD send a CANCEL\_PUSH frame when it determines that promise will not be fulfilled. If a stream has already been opened, the server can abort sending on the stream with an error code of H3\_REQUEST\_CANCELLED.

Sending a CANCEL\_PUSH frame has no direct effect on the state of existing push streams. A client SHOULD NOT send a CANCEL\_PUSH frame when it has already received a corresponding push stream. A push stream could arrive after a client has sent a CANCEL\_PUSH frame, because a server might not have processed the CANCEL\_PUSH. The client SHOULD abort reading the stream with an error code of H3\_REQUEST\_CANCELLED.

A CANCEL\_PUSH frame is sent on the control stream. Receiving a CANCEL\_PUSH frame on a stream other than the control stream MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED.

```
CANCEL_PUSH Frame {  
    Type (i) = 0x3,  
    Length (i),  
    Push ID (i),  
}
```

Figure 6: CANCEL\_PUSH Frame

The CANCEL\_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled; see Section 4.4. If a CANCEL\_PUSH frame is received that references a Push ID greater than currently allowed on the connection, this MUST be treated as a connection error of type H3\_ID\_ERROR.

If the client receives a CANCEL\_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH\_PROMISE frame due to reordering. If a server receives a CANCEL\_PUSH frame for a Push ID that has not yet been mentioned by a PUSH\_PROMISE frame, this MUST be treated as a connection error of type H3\_ID\_ERROR.

#### 7.2.4. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. Individually, a SETTINGS parameter can also be referred to as a "setting"; the identifier and value of each setting parameter can be referred to as a "setting identifier" and a "setting value".

SETTINGS frames always apply to an entire HTTP/3 connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of each control stream (see Section 6.2.1) by each peer, and MUST NOT be sent subsequently. If an endpoint receives a second SETTINGS frame on the control stream, the endpoint MUST respond with a connection error of type H3\_FRAME\_UNEXPECTED.

SETTINGS frames MUST NOT be sent on any stream other than the control stream. If an endpoint receives a SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type H3\_FRAME\_UNEXPECTED.

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer that can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - each peer uses SETTINGS to advertise a set of supported values. The definition of the setting would describe how each peer combines the two sets to conclude which choice will be used. SETTINGS does not provide a mechanism to identify when the choice takes effect.

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume a very large response field section, while servers are more cautious about request size.

The same setting identifier MUST NOT occur more than once in the SETTINGS frame. A receiver MAY treat the presence of duplicate setting identifiers as a connection error of type H3\_SETTINGS\_ERROR.

The payload of a SETTINGS frame consists of zero or more parameters. Each parameter consists of a setting identifier and a value, both encoded as QUIC variable-length integers.

```
Setting {  
    Identifier (i),  
    Value (i),  
}  
  
SETTINGS Frame {  
    Type (i) = 0x4,  
    Length (i),  
    Setting (...) ...,  
}
```

Figure 7: SETTINGS Frame

An implementation MUST ignore any parameter with an identifier it does not understand.

#### 7.2.4.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/3:

**SETTINGS\_MAX\_FIELD\_SECTION\_SIZE (0x6):** The default value is unlimited. See Section 4.1.1.3 for usage.

Setting identifiers of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown identifiers be ignored. Such settings have no defined meaning. Endpoints SHOULD include at least one such setting in their SETTINGS frame. Endpoints MUST NOT consider such settings to have any meaning upon receipt.

Because the setting has no defined meaning, the value of the setting can be any value the implementation selects.

Setting identifiers which were defined in [HTTP2] where there is no corresponding HTTP/3 setting have also been reserved (Section 11.2.2). These reserved settings MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3\_SETTINGS\_ERROR.

Additional settings can be defined by extensions to HTTP/3; see Section 9 for more details.

#### 7.2.4.2. Initialization

An HTTP implementation MUST NOT send frames or requests that would be invalid based on its current understanding of the peer's settings.

All settings begin at an initial value. Each endpoint SHOULD use these initial values to send messages before the peer's SETTINGS frame has arrived, as packets carrying the settings can be lost or delayed. When the SETTINGS frame arrives, any settings are changed to their new values.

This removes the need to wait for the SETTINGS frame before sending messages. Endpoints MUST NOT require any data to be received from the peer prior to sending the SETTINGS frame; settings MUST be sent as soon as the transport is ready to send data.

For servers, the initial value of each client setting is the default value.

For clients using a 1-RTT QUIC connection, the initial value of each server setting is the default value. 1-RTT keys will always become available prior to the packet containing SETTINGS being processed by QUIC, even if the server sends SETTINGS immediately. Clients SHOULD NOT wait indefinitely for SETTINGS to arrive before sending requests, but SHOULD process received datagrams in order to increase the likelihood of processing SETTINGS before sending the first request.

When a 0-RTT QUIC connection is being used, the initial value of each server setting is the value used in the previous session. Clients SHOULD store the settings the server provided in the HTTP/3 connection where resumption information was provided, but MAY opt not to store settings in certain cases (e.g., if the session ticket is received before the SETTINGS frame). A client MUST comply with stored settings -- or default values, if no values are stored -- when attempting 0-RTT. Once a server has provided new settings, clients MUST comply with those values.

A server can remember the settings that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the HTTP/3 settings values in determining whether to accept 0-RTT data. If the server cannot determine that the settings remembered by a client are compatible with its current settings, it MUST NOT accept 0-RTT data. Remembered settings are compatible if a client complying with those settings would not violate the server's current settings.

A server MAY accept 0-RTT and subsequently provide different settings in its SETTINGS frame. If 0-RTT data is accepted by the server, its SETTINGS frame MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. The server MUST include all settings that differ from their default values. If a server accepts 0-RTT but then sends settings that are not compatible with the previously specified settings, this MUST be treated as a

connection error of type `H3_SETTINGS_ERROR`. If a server accepts 0-RTT but then sends a `SETTINGS` frame that omits a setting value that the client understands (apart from reserved setting identifiers) that was previously specified to have a non-default value, this **MUST** be treated as a connection error of type `H3_SETTINGS_ERROR`.

#### 7.2.5. `PUSH_PROMISE`

The `PUSH_PROMISE` frame (type=0x5) is used to carry a promised request header section from server to client on a request stream, as in HTTP/2.

```
PUSH_PROMISE Frame {  
    Type (i) = 0x5,  
    Length (i),  
    Push ID (i),  
    Encoded Field Section (..),  
}
```

Figure 8: `PUSH_PROMISE` Frame

The payload consists of:

**Push ID:** A variable-length integer that identifies the server push operation. A Push ID is used in push stream headers (Section 4.4) and `CANCEL_PUSH` frames (Section 7.2.3).

**Encoded Field Section:** QPACK-encoded request header fields for the promised response. See [QPACK] for more details.

A server **MUST NOT** use a Push ID that is larger than the client has provided in a `MAX_PUSH_ID` frame (Section 7.2.7). A client **MUST** treat receipt of a `PUSH_PROMISE` frame that contains a larger Push ID than the client has advertised as a connection error of `H3_ID_ERROR`.

A server **MAY** use the same Push ID in multiple `PUSH_PROMISE` frames. If so, the decompressed request header sets **MUST** contain the same fields in the same order, and both the name and the value in each field **MUST** be exact matches. Clients **SHOULD** compare the request header sections for resources promised multiple times. If a client receives a Push ID that has already been promised and detects a mismatch, it **MUST** respond with a connection error of type `H3_GENERAL_PROTOCOL_ERROR`. If the decompressed field sections match exactly, the client **SHOULD** associate the pushed content with each stream on which a `PUSH_PROMISE` frame was received.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH\_PROMISE frame that uses a Push ID that they have already consumed and discarded are forced to ignore the promise.

If a PUSH\_PROMISE frame is received on the control stream, the client MUST respond with a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

A client MUST NOT send a PUSH\_PROMISE frame. A server MUST treat the receipt of a PUSH\_PROMISE frame as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

See Section 4.4 for a description of the overall server push mechanism.

#### 7.2.6. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of an HTTP/3 connection by either endpoint. GOAWAY allows an endpoint to stop accepting new requests or pushes while still finishing processing of previously received requests and pushes. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

```
GOAWAY Frame {  
    Type (i) = 0x7,  
    Length (i),  
    Stream ID/Push ID (...),  
}
```

Figure 9: GOAWAY Frame

The GOAWAY frame is always sent on the control stream. In the server to client direction, it carries a QUIC Stream ID for a client-initiated bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type H3\_ID\_ERROR.

In the client to server direction, the GOAWAY frame carries a Push ID encoded as a variable-length integer.

The GOAWAY frame applies to the entire connection, not a specific stream. A client **MUST** treat a GOAWAY frame on a stream other than the control stream as a connection error of type `H3_FRAME_UNEXPECTED`; see Section 8.

See Section 5.2 for more information on the use of the GOAWAY frame.

#### 7.2.7. MAX\_PUSH\_ID

The `MAX_PUSH_ID` frame (type=0xd) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in `PUSH_PROMISE` and `CANCEL_PUSH` frames. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit maintained by the QUIC transport.

The `MAX_PUSH_ID` frame is always sent on the control stream. Receipt of a `MAX_PUSH_ID` frame on any other stream **MUST** be treated as a connection error of type `H3_FRAME_UNEXPECTED`.

A server **MUST NOT** send a `MAX_PUSH_ID` frame. A client **MUST** treat the receipt of a `MAX_PUSH_ID` frame as a connection error of type `H3_FRAME_UNEXPECTED`.

The maximum Push ID is unset when an HTTP/3 connection is created, meaning that a server cannot push until it receives a `MAX_PUSH_ID` frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending `MAX_PUSH_ID` frames as the server fulfills or cancels server pushes.

```
MAX_PUSH_ID Frame {  
    Type (i) = 0xd,  
    Length (i),  
    Push ID (i),  
}
```

Figure 10: `MAX_PUSH_ID` Frame

The `MAX_PUSH_ID` frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use; see Section 4.4. A `MAX_PUSH_ID` frame cannot reduce the maximum Push ID; receipt of a `MAX_PUSH_ID` frame that contains a smaller value than previously received **MUST** be treated as a connection error of type `H3_ID_ERROR`.



### 7.2.8. Reserved Frame Types

Frame types of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored (Section 9). These frames have no semantics, and MAY be sent on any stream where frames are allowed to be sent. This enables their use for application-layer padding. Endpoints MUST NOT consider these frames to have any meaning upon receipt.

The payload and length of the frames are selected in any manner the implementation chooses.

Frame types that were used in HTTP/2 where there is no corresponding HTTP/3 frame have also been reserved (Section 11.2.1). These frame types MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED.

## 8. Error Handling

When a stream cannot be completed successfully, QUIC allows the application to abruptly terminate (reset) that stream and communicate a reason; see Section 2.4 of [QUIC-TRANSPORT]. This is referred to as a "stream error." An HTTP/3 implementation can decide to close a QUIC stream and communicate the type of error. Wire encodings of error codes are defined in Section 8.1. Stream errors are distinct from HTTP status codes which indicate error conditions. Stream errors indicate that the sender did not transfer or consume the full request or response, while HTTP status codes indicate the result of a request that was successfully received.

If an entire connection needs to be terminated, QUIC similarly provides mechanisms to communicate a reason; see Section 5.3 of [QUIC-TRANSPORT]. This is referred to as a "connection error." Similar to stream errors, an HTTP/3 implementation can terminate a QUIC connection and communicate the reason using an error code from Section 8.1.

Although the reasons for closing streams and connections are called "errors," these actions do not necessarily indicate a problem with the connection or either implementation. For example, a stream can be reset if the requested resource is no longer needed.

An endpoint MAY choose to treat a stream error as a connection error under certain circumstances, closing the entire connection in response to a condition on a single stream. Implementations need to consider the impact on outstanding requests before making this choice.

Because new error codes can be defined without negotiation (see Section 9), use of an error code in an unexpected context or receipt of an unknown error code MUST be treated as equivalent to H3\_NO\_ERROR. However, closing a stream can have other effects regardless of the error code; for example, see Section 4.1.

### 8.1. HTTP/3 Error Codes

The following error codes are defined for use when abruptly terminating streams, aborting reading of streams, or immediately closing HTTP/3 connections.

H3\_NO\_ERROR (0x100): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

H3\_GENERAL\_PROTOCOL\_ERROR (0x101): Peer violated protocol requirements in a way that does not match a more specific error code, or endpoint declines to use the more specific error code.

H3\_INTERNAL\_ERROR (0x102): An internal error has occurred in the HTTP stack.

H3\_STREAM\_CREATION\_ERROR (0x103): The endpoint detected that its peer created a stream that it will not accept.

H3\_CLOSED\_CRITICAL\_STREAM (0x104): A stream required by the HTTP/3 connection was closed or reset.

H3\_FRAME\_UNEXPECTED (0x105): A frame was received that was not permitted in the current state or on the current stream.

H3\_FRAME\_ERROR (0x106): A frame that fails to satisfy layout requirements or with an invalid size was received.

H3\_EXCESSIVE\_LOAD (0x107): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

H3\_ID\_ERROR (0x108): A Stream ID or Push ID was used incorrectly, such as exceeding a limit, reducing a limit, or being reused.

H3\_SETTINGS\_ERROR (0x109): An endpoint detected an error in the payload of a SETTINGS frame.

H3\_MISSING\_SETTINGS (0x10a): No SETTINGS frame was received at the beginning of the control stream.

H3\_REQUEST\_REJECTED (0x10b): A server rejected a request without performing any application processing.

H3\_REQUEST\_CANCELLED (0x10c): The request or its response (including pushed response) is cancelled.

H3\_REQUEST\_INCOMPLETE (0x10d): The client's stream terminated without containing a fully-formed request.

H3\_MESSAGE\_ERROR (0x10e): An HTTP message was malformed and cannot be processed.

H3\_CONNECT\_ERROR (0x10f): The TCP connection established in response to a CONNECT request was reset or abnormally closed.

H3\_VERSION\_FALLBACK (0x110): The requested operation cannot be served over HTTP/3. The peer should retry over HTTP/1.1.

Error codes of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown error codes be treated as equivalent to H3\_NO\_ERROR (Section 9). Implementations SHOULD select an error code from this space with some probability when they would have sent H3\_NO\_ERROR.

## 9. Extensions to HTTP/3

HTTP/3 permits extension of the protocol. Within the limitations described in this section, protocol extensions can be used to provide additional services or alter any aspect of the protocol. Extensions are effective only within the scope of a single HTTP/3 connection.

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or fields.

Extensions are permitted to use new frame types (Section 7.2), new settings (Section 7.2.4.1), new error codes (Section 8), or new unidirectional stream types (Section 6.2). Registries are established for managing these extension points: frame types (Section 11.2.1), settings (Section 11.2.2), error codes (Section 11.2.3), and stream types (Section 11.2.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames and abort reading on unidirectional streams that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, where a known frame type is required to be in a specific location, such as the SETTINGS frame as the first frame of the control stream (see Section 6.2.1), an unknown frame type does not satisfy that requirement and SHOULD be treated as an error.

Extensions that could change the semantics of existing protocol components **MUST** be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. Coordinating when such a revised layout comes into effect could prove complex. As such, allocating new identifiers for new definitions of existing protocol elements is likely to be more effective.

This document does not mandate a specific method for negotiating the use of an extension but notes that a setting (Section 7.2.4.1) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the default value **MUST** be defined in such a fashion that the extension is disabled if the setting is omitted.

## 10. Security Considerations

The security considerations of HTTP/3 should be comparable to those of HTTP/2 with TLS. However, many of the considerations from Section 10 of [HTTP2] apply to [QUIC-TRANSPORT] and are discussed in that document.

### 10.1. Server Authority

HTTP/3 relies on the HTTP definition of authority. The security considerations of establishing authority are discussed in Section 17.1 of [SEMANTICS].

### 10.2. Cross-Protocol Attacks

The use of ALPN in the TLS and QUIC handshakes establishes the target application protocol before application-layer bytes are processed. This ensures that endpoints have strong assurances that peers are using the same protocol.

This does not guarantee protection from all cross-protocol attacks. Section 21.5 of [QUIC-TRANSPORT] describes some ways in which the plaintext of QUIC packets can be used to perform request forgery against endpoints that don't use authenticated transports.

### 10.3. Intermediary Encapsulation Attacks

The HTTP/3 field encoding allows the expression of names that are not valid field names in the syntax used by HTTP (Section 5.1 of [SEMANTICS]). Requests or responses containing invalid field names MUST be treated as malformed (Section 4.1.3). An intermediary therefore cannot translate an HTTP/3 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/3 can transport field values that are not valid. While most values that can be encoded will not alter field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a field value MUST be treated as malformed (Section 4.1.3). Valid characters are defined by the "field-content" ABNF rule in Section 5.5 of [SEMANTICS].

### 10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH\_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server MUST ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Clients are required to reject pushed responses for which an origin server is not authoritative; see Section 4.4.

### 10.5. Denial-of-Service Considerations

An HTTP/3 connection can demand a greater commitment of resources to operate than an HTTP/1.1 or HTTP/2 connection. The use of field compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH\_PROMISE frames is constrained in a similar fashion. A client that accepts server push SHOULD limit the number of Push IDs it issues at a time.

Processing capacity cannot be guarded as effectively as state capacity.

The ability to send undefined protocol elements that the peer is required to ignore can be abused to cause a peer to expend additional processing time. This might be done by setting multiple undefined SETTINGS parameters, unknown frame types, or unknown stream types. Note, however, that some uses are entirely legitimate, such as optional-to-understand extensions and padding to increase resistance to traffic analysis.

Compression of field sections also offers some opportunities to waste processing resources; see Section 7 of [QPACK] for more details on potential abuses.

All these features -- i.e., server push, unknown protocol elements, field compression -- have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that does not monitor such behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error of type H3\_EXCESSIVE\_LOAD (Section 8), but false positives will result in disrupting valid connections and requests.

#### 10.5.1. Limits on Field Section Size

A large field section (Section 4.1) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header section, which prevents streaming of the header section to its ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean that an endpoint likely needs to buffer the entire header section. Since there is no hard limit to the size of a field section, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the SETTINGS\_MAX\_FIELD\_SECTION\_SIZE (Section 4.1.1.3) setting to advise peers of limits that might apply on the size of field sections. This setting is only advisory, so endpoints MAY choose to send field sections that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to an HTTP/3 connection, so any request or

response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger field section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process.

#### 10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on a proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. Therefore, a proxy that supports CONNECT might be more conservative in the number of simultaneous requests it accepts.

A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME\_WAIT state. To account for this, a proxy might delay increasing the QUIC stream limits for some time after a TCP connection terminates.

#### 10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/3 enables compression of fields (Section 4.1.1); the following concerns also apply to the use of HTTP compressed content-codings; see Section 8.4.1 of [SEMANTICS].

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression contexts are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined.

Further considerations regarding the compression of field sections are described in [QPACK].

### 10.7. Padding and Traffic Analysis

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Where HTTP/2 employs PADDING frames and Padding fields in other frames to make a connection more resistant to traffic analysis, HTTP/3 can either rely on transport-layer padding or employ the reserved frame and stream types discussed in Section 7.2.8 and Section 6.2.3. These methods of padding produce different results in terms of the granularity of padding, how padding is arranged in relation to the information that is being protected, whether padding is applied in the case of packet loss, and how an implementation might control padding.

Reserved stream types can be used to give the appearance of sending traffic even when the connection is idle. Because HTTP traffic often occurs in bursts, apparent traffic can be used to obscure the timing or duration of such bursts, even to the point of appearing to send a constant stream of data. However, as such traffic is still flow controlled by the receiver, a failure to promptly drain such streams and provide additional flow control credit can limit the sender's ability to send real traffic.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Use of padding can result in less protection than might seem immediately obvious. Redundant padding could even be counterproductive. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

### 10.8. Frame Parsing

Several protocol elements contain nested length elements, typically in the form of frames with an explicit length containing variable-length integers. This could pose a security risk to an incautious implementer. An implementation **MUST** ensure that the length of a frame exactly matches the length of the fields it contains.



### 10.9. Early Data

The use of 0-RTT with HTTP/3 creates an exposure to replay attack. The anti-replay mitigations in [HTTP-REPLAY] MUST be applied when using HTTP/3 with 0-RTT. When applying [HTTP-REPLAY] to HTTP/3, references to the TLS layer refer to the handshake performed within QUIC, while all references to application data refer to the contents of streams.

### 10.10. Migration

Certain HTTP implementations use the client address for logging or access-control purposes. Since a QUIC client's address might change during a connection (and future versions might support simultaneous use of multiple addresses), such implementations will need to either actively retrieve the client's current address or addresses when they are relevant or explicitly accept that the original address might change.

### 10.11. Privacy Considerations

Several characteristics of HTTP/3 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client.

HTTP/3's preference for using a single QUIC connection allows correlation of a user's activity on a site. Reusing connections for different origins allows for correlation of activity across those origins.

Several features of QUIC solicit immediate responses and can be used by an endpoint to measure latency to their peer; this might have privacy implications in certain scenarios.

## 11. IANA Considerations

This document registers a new ALPN protocol ID (Section 11.1) and creates new registries that manage the assignment of codepoints in HTTP/3.

### 11.1. Registration of HTTP/3 Identification String

This document creates a new registration for the identification of HTTP/3 in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "h3" string identifies HTTP/3:

Protocol: HTTP/3

Identification Sequence: 0x68 0x33 ("h3")

Specification: This document

### 11.2. New Registries

New registries created in this document operate under the QUIC registration policy documented in Section 22.1 of [QUIC-TRANSPORT]. These registries all include the common set of fields listed in Section 22.1.1 of [QUIC-TRANSPORT]. These registries [SHALL be/are] collected under a "Hypertext Transfer Protocol version 3 (HTTP/3) Parameters" heading.

The initial allocations in these registries created in this document are all assigned permanent status and list a change controller of the IETF and a contact of the HTTP working group (ietf-http-wg@w3.org).

#### 11.2.1. Frame Types

This document establishes a registry for HTTP/3 frame type codes. The "HTTP/3 Frame Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [HTTP2], it is preferable that the assignments parallel each other where the code spaces overlap. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following field:

Frame Type: A name or label for the frame type.

Specifications of frame types MUST include a description of the frame layout and its semantics, including any parts of the frame that are conditionally present.

The entries in Table 2 are registered by this document.

Frame Type	Value	Specification
DATA	0x0	Section 7.2.1
HEADERS	0x1	Section 7.2.2
Reserved	0x2	N/A
CANCEL_PUSH	0x3	Section 7.2.3
SETTINGS	0x4	Section 7.2.4
PUSH_PROMISE	0x5	Section 7.2.5
Reserved	0x6	N/A
GOAWAY	0x7	Section 7.2.6
Reserved	0x8	N/A
Reserved	0x9	N/A
MAX_PUSH_ID	0xd	Section 7.2.7

Table 2: Initial HTTP/3 Frame Types

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

## 11.2.2. Settings Parameters

This document establishes a registry for HTTP/3 settings. The "HTTP/3 Settings" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Settings" registry defined in [HTTP2], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

**Setting Name:** A symbolic name for the setting. Specifying a setting name is optional.

**Default:** The value of the setting unless otherwise indicated. A default SHOULD be the most restrictive possible value.

The entries in Table 3 are registered by this document.

Setting Name	Value	Specification	Default
Reserved	0x0	N/A	N/A
Reserved	0x2	N/A	N/A
Reserved	0x3	N/A	N/A
Reserved	0x4	N/A	N/A
Reserved	0x5	N/A	N/A
MAX_FIELD_SECTION_SIZE	0x6	Section 7.2.4.1	Unlimited

Table 3: Initial HTTP/3 Settings

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

### 11.2.3. Error Codes

This document establishes a registry for HTTP/3 error codes. The "HTTP/3 Error Code" registry manages a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated. Use of values that are registered in the "HTTP/2 Error Code" registry is discouraged, and expert reviewers MAY reject such registrations.

In addition to common fields as described in Section 11.2, this registry includes two additional fields. Permanent registrations in this registry MUST include the following field:

Name: A name for the error code.

Description: A brief description of the error code semantics.

The entries in Table 4 are registered by this document. These error codes were selected from the range that operates on a Specification Required policy to avoid collisions with HTTP/2 error codes.

Name	Value	Description	Specification
H3_NO_ERROR	0x100	No error	Section 8.1
H3_GENERAL_PROTOCOL_ERROR	0x101	General protocol error	Section 8.1
H3_INTERNAL_ERROR	0x102	Internal error	Section 8.1
H3_STREAM_CREATION_ERROR	0x103	Stream	Section 8.1

		creation error	
H3_CLOSED_CRITICAL_STREAM	0x104	Critical stream was closed	Section 8.1
H3_FRAME_UNEXPECTED	0x105	Frame not permitted in the current state	Section 8.1
H3_FRAME_ERROR	0x106	Frame violated layout or size rules	Section 8.1
H3_EXCESSIVE_LOAD	0x107	Peer generating excessive load	Section 8.1
H3_ID_ERROR	0x108	An identifier was used incorrectly	Section 8.1
H3_SETTINGS_ERROR	0x109	SETTINGS frame contained invalid values	Section 8.1
H3_MISSING_SETTINGS	0x10a	No SETTINGS frame received	Section 8.1
H3_REQUEST_REJECTED	0x10b	Request not processed	Section 8.1
H3_REQUEST_CANCELLED	0x10c	Data no longer needed	Section 8.1
H3_REQUEST_INCOMPLETE	0x10d	Stream terminated early	Section 8.1

H3_MESSAGE_ERROR	0x10e	Malformed message	Section 8.1
H3_CONNECT_ERROR	0x10f	TCP reset or error on CONNECT request	Section 8.1
H3_VERSION_FALLBACK	0x110	Retry over HTTP/1.1	Section 8.1

Table 4: Initial HTTP/3 Error Codes

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

#### 11.2.4. Stream Types

This document establishes a registry for HTTP/3 unidirectional stream types. The "HTTP/3 Stream Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

**Stream Type:** A name or label for the stream type.

**Sender:** Which endpoint on an HTTP/3 connection may initiate a stream of this type. Values are "Client", "Server", or "Both".

Specifications for permanent registrations MUST include a description of the stream type, including the layout and semantics of the stream contents.

The entries in the following table are registered by this document.

Stream Type	Value	Specification	Sender
Control Stream	0x00	Section 6.2.1	Both
Push Stream	0x01	Section 4.4	Server

Table 5

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

## 12. References

### 12.1. Normative References

- [ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.
- [CACHING] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-14.txt>>.
- [HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/info/rfc8470>>.
- [QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-21, 2 February 2021, <<https://tools.ietf.org/html/draft-ietf-quic-qpack-21>>.
- [QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 2 February 2021, <<https://tools.ietf.org/html/draft-ietf-quic-transport-34>>.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.



- [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/rfc2119>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [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/rfc8126>>.
- [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/rfc8174>>.
- [SEMANTICS] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-semantics-14.txt>>.
- [URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.

## 12.2. Informative References

- [BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <<http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>>.

## [DNS-TERMS]

Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <<https://www.rfc-editor.org/info/rfc8499>>.

## [HPACK]

Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<https://www.rfc-editor.org/info/rfc7541>>.

## [HTTP11]

Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-messaging-14.txt>>.

## [HTTP2]

Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

## [RFC6585]

Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <<https://www.rfc-editor.org/info/rfc6585>>.

## [RFC8164]

Nottingham, M. and M. Thomson, "Opportunistic Security for HTTP/2", RFC 8164, DOI 10.17487/RFC8164, May 2017, <<https://www.rfc-editor.org/info/rfc8164>>.

## [TFO]

Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

## [TLS13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

## Appendix A. Considerations for Transitioning from HTTP/2

HTTP/3 is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/3, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/3.

HTTP/3 begins from the premise that similarity to HTTP/2 is preferable, but not a hard requirement. HTTP/3 departs from HTTP/2 where QUIC differs from TCP, either to take advantage of QUIC features (like streams) or to accommodate important shortcomings (such as a lack of total ordering). These differences make HTTP/3 similar to HTTP/2 in key aspects, such as the relationship of requests and responses to streams. However, the details of the HTTP/3 design are substantially different from HTTP/2.

Some important departures are noted in this section.

#### A.1. Streams

HTTP/3 permits use of a larger number of streams ( $2^{62}-1$ ) than HTTP/2. The same considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

In contrast to HTTP/2, stream concurrency in HTTP/3 is managed by QUIC. QUIC considers a stream closed when all data has been received and sent data has been acknowledged by the peer. HTTP/2 considers a stream closed when the frame containing the END\_STREAM bit has been committed to the transport. As a result, the stream for an equivalent exchange could remain "active" for a longer period of time. HTTP/3 servers might choose to permit a larger number of concurrent client-initiated bidirectional streams to achieve equivalent concurrency to HTTP/2, depending on the expected usage patterns.

In HTTP/2, only request and response bodies (the frame payload of DATA frames) are subject to flow control. All HTTP/3 frames are sent on QUIC streams, so all frames on all streams are flow-controlled in HTTP/3.

Due to the presence of other unidirectional stream types, HTTP/3 does not rely exclusively on the number of concurrent unidirectional streams to control the number of concurrent in-flight pushes. Instead, HTTP/3 clients use the MAX\_PUSH\_ID frame to control the number of pushes received from an HTTP/3 server.

## A.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an `END_STREAM` flag is not required. This permits the removal of the `Flags` field from the generic frame layout.

Frame payloads are largely drawn from [HTTP2]. However, QUIC includes many features (e.g., flow control) that are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/3. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/3 implementations. However, even frame types that appear in both mappings do not have identical semantics.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/3 will break them.

Some examples of feature adaptations are described below, as well as general guidance to extension frame implementors converting an HTTP/2 extension to HTTP/3.

### A.2.1. Prioritization Differences

HTTP/2 specifies priority assignments in `PRIORITY` frames and (optionally) in `HEADERS` frames. HTTP/3 does not provide a means of signaling priority.

Note that while there is no explicit signaling for priority, this does not mean that prioritization is not important for achieving good performance.

### A.2.2. Field Compression Differences

HPACK was designed with the assumption of in-order delivery. A sequence of encoded field sections must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

Because this total ordering is not provided by QUIC, HTTP/3 uses a modified version of HPACK, called QPACK. QPACK uses a single unidirectional stream to make all modifications to the dynamic table, ensuring a total order of updates. All frames that contain encoded fields merely reference the table state at a given time without modifying it.

[QPACK] provides additional details.

#### A.2.3. Flow Control Differences

HTTP/2 specifies a stream flow control mechanism. Although all HTTP/2 frames are delivered on streams, only the DATA frame payload is subject to flow control. QUIC provides flow control for stream data and all HTTP/3 frame types defined in this document are sent on streams. Therefore, all frame headers and payload are subject to flow control.

#### A.2.4. Guidance for New Frame Type Definitions

Frame type definitions in HTTP/3 often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allows for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/3 use an identifier other than a Stream ID (e.g., Push IDs). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Because the Flags field is not present in generic HTTP/3 frames, those frames that depend on the presence of flags need to allocate space for flags as part of their frame payload.

Other than these issues, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with a control stream in HTTP/3. HTTP/3 extensions will not assume ordering, but would not be harmed by ordering, and are expected to be portable to HTTP/2.

#### A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types

DATA (0x0): Padding is not defined in HTTP/3 frames. See Section 7.2.1.

HEADERS (0x1): The PRIORITY region of HEADERS is not defined in HTTP/3 frames. Padding is not defined in HTTP/3 frames. See Section 7.2.2.

PRIORITY (0x2): As described in Appendix A.2.1, HTTP/3 does not

provide a means of signaling priority.

**RST\_STREAM (0x3):** RST\_STREAM frames do not exist in HTTP/3, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL\_PUSH frame (Section 7.2.3).

**SETTINGS (0x4):** SETTINGS frames are sent only at the beginning of the connection. See Section 7.2.4 and Appendix A.3.

**PUSH\_PROMISE (0x5):** The PUSH\_PROMISE frame does not reference a stream; instead the push stream references the PUSH\_PROMISE frame using a Push ID. See Section 7.2.5.

**PING (0x6):** PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.

**GOAWAY (0x7):** GOAWAY does not contain an error code. In the client to server direction, it carries a Push ID instead of a server initiated stream ID. See Section 7.2.6.

**WINDOW\_UPDATE (0x8):** WINDOW\_UPDATE frames do not exist in HTTP/3, since QUIC provides flow control.

**CONTINUATION (0x9):** CONTINUATION frames do not exist in HTTP/3; instead, larger HEADERS/PUSH\_PROMISE frames than HTTP/2 are permitted.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/3 if still applicable. The IDs of frames defined in [HTTP2] have been reserved for simplicity. Note that the frame type space in HTTP/3 is substantially larger (62 bits versus 8 bits), so many HTTP/3 frame types have no equivalent HTTP/2 code points. See Section 11.2.1.

### A.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, as the first frame of the control stream, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/3. The HTTP-level setting that is retained in HTTP/3 has the same value as in HTTP/2. The superseded settings are reserved, and their receipt is an error. See Section 7.2.4.1 for discussion of both the retained and reserved values.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS\_HEADER\_TABLE\_SIZE (0x1): See [QPACK].

SETTINGS\_ENABLE\_PUSH (0x2): This is removed in favor of the MAX\_PUSH\_ID frame, which provides a more granular control over server push. Specifying a setting with the identifier 0x2 (corresponding to the SETTINGS\_ENABLE\_PUSH parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_MAX\_CONCURRENT\_STREAMS (0x3): QUIC controls the largest open Stream ID as part of its flow control logic. Specifying a setting with the identifier 0x3 (corresponding to the SETTINGS\_MAX\_CONCURRENT\_STREAMS parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_INITIAL\_WINDOW\_SIZE (0x4): QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying a setting with the identifier 0x4 (corresponding to the SETTINGS\_INITIAL\_WINDOW\_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_MAX\_FRAME\_SIZE (0x5): This setting has no equivalent in HTTP/3. Specifying a setting with the identifier 0x5 (corresponding to the SETTINGS\_MAX\_FRAME\_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_MAX\_HEADER\_LIST\_SIZE (0x6): This setting identifier has been renamed SETTINGS\_MAX\_FIELD\_SECTION\_SIZE.

In HTTP/3, setting values are variable-length integers (6, 14, 30, or 62 bits long) rather than fixed-length 32-bit fields as in HTTP/2. This will often produce a shorter encoding, but can produce a longer encoding for settings that use the full 32-bit space. Settings ported from HTTP/2 might choose to redefine their value to limit it to 30 bits for more efficient encoding, or to make use of the 62-bit space if more than 30 bits are required.

Settings need to be defined separately for HTTP/2 and HTTP/3. The IDs of settings defined in [HTTP2] have been reserved for simplicity. Note that the settings identifier space in HTTP/3 is substantially larger (62 bits versus 16 bits), so many HTTP/3 settings have no equivalent HTTP/2 code point. See Section 11.2.2.

As QUIC streams might arrive out of order, endpoints are advised not to wait for the peers' settings to arrive before responding to other streams. See Section 7.2.4.2.

#### A.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, the differences between HTTP/2 and HTTP/3 mean that error codes are not directly portable between versions.

The HTTP/2 error codes defined in Section 7 of [HTTP2] logically map to the HTTP/3 error codes as follows:

NO\_ERROR (0x0): H3\_NO\_ERROR in Section 8.1.

PROTOCOL\_ERROR (0x1): This is mapped to H3\_GENERAL\_PROTOCOL\_ERROR except in cases where more specific error codes have been defined. Such cases include H3\_FRAME\_UNEXPECTED, H3\_MESSAGE\_ERROR, and H3\_CLOSED\_CRITICAL\_STREAM defined in Section 8.1.

INTERNAL\_ERROR (0x2): H3\_INTERNAL\_ERROR in Section 8.1.

FLOW\_CONTROL\_ERROR (0x3): Not applicable, since QUIC handles flow control.

SETTINGS\_TIMEOUT (0x4): Not applicable, since no acknowledgment of SETTINGS is defined.

STREAM\_CLOSED (0x5): Not applicable, since QUIC handles stream management.

FRAME\_SIZE\_ERROR (0x6): H3\_FRAME\_ERROR error code defined in Section 8.1.

REFUSED\_STREAM (0x7): H3\_REQUEST\_REJECTED (in Section 8.1) is used to indicate that a request was not processed. Otherwise, not applicable because QUIC handles stream management.

CANCEL (0x8): H3\_REQUEST\_CANCELLED in Section 8.1.

COMPRESSION\_ERROR (0x9): Multiple error codes are defined in [QPACK].

CONNECT\_ERROR (0xa): H3\_CONNECT\_ERROR in Section 8.1.

ENHANCE\_YOUR\_CALM (0xb): H3\_EXCESSIVE\_LOAD in Section 8.1.

INADEQUATE\_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP\_1\_1\_REQUIRED (0xd): H3\_VERSION\_FALLBACK in Section 8.1.



Error codes need to be defined for HTTP/2 and HTTP/3 separately. See Section 11.2.3.

#### A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors

An intermediary that converts between HTTP/2 and HTTP/3 may encounter error conditions from either upstream. It is useful to communicate the occurrence of error to the downstream but error codes largely reflect connection-local problems that generally do not make sense to propagate.

An intermediary that encounters an error from an upstream origin can indicate this by sending an HTTP status code such as 502, which is suitable for a broad class of errors.

There are some rare cases where it is beneficial to propagate the error by mapping it to the closest matching error type to the receiver. For example, an intermediary that receives an HTTP/2 stream error of type `REFUSED_STREAM` from the origin has a clear signal that the request was not processed and that the request is safe to retry. Propagating this error condition to the client as an HTTP/3 stream error of type `H3_REQUEST_REJECTED` allows the client to take the action it deems most appropriate. In the reverse direction, the intermediary might deem it beneficial to pass on client request cancellations that are indicated by terminating a stream with `H3_REQUEST_CANCELLED`; see Section 4.1.2.

Conversion between errors is described in the logical mapping. The error codes are defined in non-overlapping spaces in order to protect against accidental conversion that could result in the use of inappropriate or unknown error codes for the target version. An intermediary is permitted to promote stream errors to connection errors but they should be aware of the cost to the HTTP/3 connection for what might be a temporary or intermittent error.

#### Appendix B. Change Log

\*RFC Editor's Note:\* Please remove this section prior to publication of a final version of this document.

##### B.1. Since draft-ietf-quic-http-32

- \* Removed draft version guidance; added final version string
- \* Added `H3_MESSAGE_ERROR` for malformed messages

## B.2. Since draft-ietf-quic-http-31

Editorial changes only.

## B.3. Since draft-ietf-quic-http-30

Editorial changes only.

## B.4. Since draft-ietf-quic-http-29

- \* Require a connection error if a reserved frame type that corresponds to a frame in HTTP/2 is received (#3991, #3993)
- \* Require a connection error if a reserved setting that corresponds to a setting in HTTP/2 is received (#3954, #3955)

## B.5. Since draft-ietf-quic-http-28

- \* CANCEL\_PUSH is recommended even when the stream is reset (#3698, #3700)
- \* Use H3\_ID\_ERROR when GOAWAY contains a larger identifier (#3631, #3634)

## B.6. Since draft-ietf-quic-http-27

- \* Updated text to refer to latest HTTP revisions
- \* Use the HTTP definition of authority for establishing and coalescing connections (#253, #2223, #3558)
- \* Define use of GOAWAY from both endpoints (#2632, #3129)
- \* Require either :authority or Host if the URI scheme has a mandatory authority component (#3408, #3475)

## B.7. Since draft-ietf-quic-http-26

- \* No changes

## B.8. Since draft-ietf-quic-http-25

- \* Require QUICv1 for HTTP/3 (#3117, #3323)
- \* Remove DUPLICATE\_PUSH and allow duplicate PUSH\_PROMISE (#3275, #3309)
- \* Clarify the definition of "malformed" (#3352, #3345)

## B.9. Since draft-ietf-quic-http-24

- \* Removed H3\_EARLY\_RESPONSE error code; H3\_NO\_ERROR is recommended instead (#3130, #3208)
- \* Unknown error codes are equivalent to H3\_NO\_ERROR (#3276, #3331)
- \* Some error codes are reserved for greasing (#3325, #3360)

## B.10. Since draft-ietf-quic-http-23

- \* Removed "quic" Alt-Svc parameter (#3061, #3118)
- \* Clients need not persist unknown settings for use in 0-RTT (#3110, #3113)
- \* Clarify error cases around CANCEL\_PUSH (#2819, #3083)

## B.11. Since draft-ietf-quic-http-22

- \* Removed priority signaling (#2922, #2924)
- \* Further changes to error codes (#2662, #2551):
  - Error codes renumbered
  - HTTP\_MALFORMED\_FRAME replaced by HTTP\_FRAME\_ERROR, HTTP\_ID\_ERROR, and others
- \* Clarify how unknown frame types interact with required frame sequence (#2867, #2858)
- \* Describe interactions with the transport in terms of defined interface terms (#2857, #2805)
- \* Require the use of the "http-opportunistic" resource (RFC 8164) when scheme is "http" (#2439, #2973)
- \* Settings identifiers cannot be duplicated (#2979)
- \* Changes to SETTINGS frames in 0-RTT (#2972, #2790, #2945):
  - Servers must send all settings with non-default values in their SETTINGS frame, even when resuming
  - If a client doesn't have settings associated with a 0-RTT ticket, it uses the defaults

- Servers can't accept early data if they cannot recover the settings the client will have remembered
- \* Clarify that Upgrade and the 101 status code are prohibited (#2898, #2889)
- \* Clarify that frame types reserved for greasing can occur on any stream, but frame types reserved due to HTTP/2 correspondence are prohibited (#2997, #2692, #2693)
- \* Unknown error codes cannot be treated as errors (#2998, #2816)

B.12. Since draft-ietf-quic-http-21

No changes

B.13. Since draft-ietf-quic-http-20

- \* Prohibit closing the control stream (#2509, #2666)
- \* Change default priority to use an orphan node (#2502, #2690)
- \* Exclusive priorities are restored (#2754, #2781)
- \* Restrict use of frames when using CONNECT (#2229, #2702)
- \* Close and maybe reset streams if a connection error occurs for CONNECT (#2228, #2703)
- \* Encourage provision of sufficient unidirectional streams for QPACK (#2100, #2529, #2762)
- \* Allow extensions to use server-initiated bidirectional streams (#2711, #2773)
- \* Clarify use of maximum header list size setting (#2516, #2774)
- \* Extensive changes to error codes and conditions of their sending
  - Require connection errors for more error conditions (#2511, #2510)
  - Updated the error codes for illegal GOAWAY frames (#2714, #2707)
  - Specified error code for HEADERS on control stream (#2708)
  - Specified error code for servers receiving PUSH\_PROMISE (#2709)

- Specified error code for receiving DATA before HEADERS (#2715)
- Describe malformed messages and their handling (#2410, #2764)
- Remove HTTP\_PUSH\_ALREADY\_IN\_CACHE error (#2812, #2813)
- Refactor Push ID related errors (#2818, #2820)
- Rationalize HTTP/3 stream creation errors (#2821, #2822)

B.14. Since draft-ietf-quic-http-19

- \* SETTINGS\_NUM\_PLACEHOLDERS is 0x9 (#2443, #2530)
- \* Non-zero bits in the Empty field of the PRIORITY frame MAY be treated as an error (#2501)

B.15. Since draft-ietf-quic-http-18

- \* Resetting streams following a GOAWAY is recommended, but not required (#2256, #2457)
- \* Use variable-length integers throughout (#2437, #2233, #2253, #2275)
  - Variable-length frame types, stream types, and settings identifiers
  - Renumbered stream type assignments
  - Modified associated reserved values
- \* Frame layout switched from Length-Type-Value to Type-Length-Value (#2395, #2235)
- \* Specified error code for servers receiving DUPLICATE\_PUSH (#2497)
- \* Use connection error for invalid PRIORITY (#2507, #2508)

B.16. Since draft-ietf-quic-http-17

- \* HTTP\_REQUEST\_REJECTED is used to indicate a request can be retried (#2106, #2325)
- \* Changed error code for GOAWAY on the wrong stream (#2231, #2343)

## B.17. Since draft-ietf-quic-http-16

- \* Rename "HTTP/QUIC" to "HTTP/3" (#1973)
- \* Changes to PRIORITY frame (#1865, #2075)
  - Permitted as first frame of request streams
  - Remove exclusive reprioritization
  - Changes to Prioritized Element Type bits
- \* Define DUPLICATE\_PUSH frame to refer to another PUSH\_PROMISE (#2072)
- \* Set defaults for settings, allow request before receiving SETTINGS (#1809, #1846, #2038)
- \* Clarify message processing rules for streams that aren't closed (#1972, #2003)
- \* Removed reservation of error code 0 and moved HTTP\_NO\_ERROR to this value (#1922)
- \* Removed prohibition of zero-length DATA frames (#2098)

## B.18. Since draft-ietf-quic-http-15

Substantial editorial reorganization; no technical changes.

## B.19. Since draft-ietf-quic-http-14

- \* Recommend sensible values for QUIC transport parameters (#1720, #1806)
- \* Define error for missing SETTINGS frame (#1697, #1808)
- \* Setting values are variable-length integers (#1556, #1807) and do not have separate maximum values (#1820)
- \* Expanded discussion of connection closure (#1599, #1717, #1712)
- \* HTTP\_VERSION\_FALLBACK falls back to HTTP/1.1 (#1677, #1685)

## B.20. Since draft-ietf-quic-http-13

- \* Reserved some frame types for grease (#1333, #1446)

- \* Unknown unidirectional stream types are tolerated, not errors; some reserved for grease (#1490, #1525)
- \* Require settings to be remembered for 0-RTT, prohibit reductions (#1541, #1641)
- \* Specify behavior for truncated requests (#1596, #1643)

B.21. Since draft-ietf-quic-http-12

- \* TLS SNI extension isn't mandatory if an alternative method is used (#1459, #1462, #1466)
- \* Removed flags from HTTP/3 frames (#1388, #1398)
- \* Reserved frame types and settings for use in preserving extensibility (#1333, #1446)
- \* Added general error code (#1391, #1397)
- \* Unidirectional streams carry a type byte and are extensible (#910, #1359)
- \* Priority mechanism now uses explicit placeholders to enable persistent structure in the tree (#441, #1421, #1422)

B.22. Since draft-ietf-quic-http-11

- \* Moved QPACK table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

B.23. Since draft-ietf-quic-http-10

- \* Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.24. Since draft-ietf-quic-http-09

- \* Selected QCRAM for header compression (#228, #1117)
- \* The server\_name TLS extension is now mandatory (#296, #495)
- \* Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.25. Since draft-ietf-quic-http-08

- \* Clarified connection coalescing rules (#940, #1024)

B.26. Since draft-ietf-quic-http-07

- \* Changes for integer encodings in QUIC (#595, #905)
- \* Use unidirectional streams as appropriate (#515, #240, #281, #886)
- \* Improvement to the description of GOAWAY (#604, #898)
- \* Improve description of server push usage (#947, #950, #957)

B.27. Since draft-ietf-quic-http-06

- \* Track changes in QUIC error code usage (#485)

B.28. Since draft-ietf-quic-http-05

- \* Made push ID sequential, add MAX\_PUSH\_ID, remove SETTINGS\_ENABLE\_PUSH (#709)
- \* Guidance about keep-alive and QUIC PINGs (#729)
- \* Expanded text on GOAWAY and cancellation (#757)

B.29. Since draft-ietf-quic-http-04

- \* Cite RFC 5234 (#404)
- \* Return to a single stream per request (#245, #557)
- \* Use separate frame type and settings registries from HTTP/2 (#81)
- \* SETTINGS\_ENABLE\_PUSH instead of SETTINGS\_DISABLE\_PUSH (#477)
- \* Restored GOAWAY (#696)
- \* Identify server push using Push ID rather than a stream ID (#702, #281)
- \* DATA frames cannot be empty (#700)

B.30. Since draft-ietf-quic-http-03

None.



B.31. Since draft-ietf-quic-http-02

- \* Track changes in transport draft

B.32. Since draft-ietf-quic-http-01

- \* SETTINGS changes (#181):
  - SETTINGS can be sent only once at the start of a connection; no changes thereafter
  - SETTINGS\_ACK removed
  - Settings can only occur in the SETTINGS frame a single time
  - Boolean format updated
- \* Alt-Svc parameter changed from "v" to "quic"; format updated (#229)
- \* Closing the connection control stream or any message control stream is a fatal error (#176)
- \* HPACK Sequence counter can wrap (#173)
- \* 0-RTT guidance added
- \* Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.33. Since draft-ietf-quic-http-00

- \* Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)
- \* Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)
- \* Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01
- \* Reworked SETTINGS\_ACK to account for indeterminate inter-stream order (#75)
- \* Described CONNECT pseudo-method (#95)
- \* Updated ALPN token and Alt-Svc guidance (#13,#87)
- \* Application-layer-defined error codes (#19,#74)

B.34. Since draft-shade-quic-http2-mapping-00

- \* Adopted as base for draft-ietf-quic-http
- \* Updated authors/editors list

#### Acknowledgments

The original authors of this specification were Robbie Shade and Mike Warres.

The IETF QUIC Working Group received an enormous amount of support from many people. Among others, the following people provided substantial contributions to this document:

- \* Bence Beky
- \* Daan De Meyer
- \* Martin Duke
- \* Roy Fielding
- \* Alan Frindell
- \* Alessandro Ghedini
- \* Nick Harper
- \* Ryan Hamilton
- \* Christian Huitema
- \* Subodh Iyengar
- \* Robin Marx
- \* Patrick McManus
- \* Luca Niccolini
- \* (Kazuho Oku)
- \* Lucas Pardue
- \* Roberto Peon
- \* Julian Reschke

- \* Eric Rescorla
- \* Martin Seemann
- \* Ben Schwartz
- \* Ian Swett
- \* Willy Taureau
- \* Martin Thomson
- \* Dmitri Tikhonov
- \* Tatsuhiro Tsujikawa

A portion of Mike's contribution was supported by Microsoft during his employment there.

Author's Address

Mike Bishop (editor)  
Akamai

Email: [mbishop@evequefou.be](mailto:mbishop@evequefou.be)

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: August 5, 2017

M. Nottingham  
February 1, 2017

Retrying HTTP Requests  
draft-nottingham-httpbis-retry-01

Abstract

HTTP allows requests to be automatically retried under certain circumstances. This draft explores how this is implemented, requirements for similar functionality from other parts of the stack, and potential future improvements.

Note to Readers

This draft is not intended to be published as an RFC.

The issues list for this draft can be found at  
<https://github.com/mnot/I-D/labels/httpbis-retry> .

The most recent (often, unpublished) draft is at  
<https://mnot.github.io/I-D/httpbis-retry/> .

Recent changes are listed at <https://github.com/mnot/I-D/commits/gh-pages/httpbis-retry> .

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 August 5, 2017.

## Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. Notational Conventions . . . . .	3
2. Background . . . . .	3
2.1. Retries and Replays: A Taxonomy of Repetition . . . . .	3
2.2. What the Spec Says: Automatic Retries . . . . .	4
2.3. What the Specs Say: Replay . . . . .	5
2.3.1. TCP Fast Open . . . . .	5
2.3.2. TLS 1.3 . . . . .	5
2.3.3. QUIC . . . . .	6
3. Discussion . . . . .	6
3.1. Automatic Retries In Practice . . . . .	6
3.2. Replays Are Different . . . . .	7
4. Possible Areas of Work . . . . .	8
4.1. Updating HTTP's Requirements for Retries . . . . .	8
4.2. Protocol Extensions . . . . .	9
4.3. Feedback to Transport ORTT Efforts . . . . .	9
5. Security Considerations . . . . .	9
6. Acknowledgements . . . . .	9
7. References . . . . .	10
7.1. Normative References . . . . .	10
7.2. Informative References . . . . .	10
7.3. URIs . . . . .	11
Appendix A. When Clients Retry . . . . .	11
A.1. Squid . . . . .	11
A.2. Traffic Server . . . . .	12
A.3. Firefox . . . . .	14
A.4. Chromium . . . . .	16
A.5. Curl . . . . .	17
Author's Address . . . . .	18

## 1. Introduction

One of the benefits of HTTP's well-defined method semantics is that they allow failed requests to be retried, under certain circumstances.

However, interest in extending, redefining or just clarifying HTTP's retry semantics is increasing, for a number of reasons:

- o Since HTTP/1.1's requirements were written, there has been a substantial amount of experience deploying and using HTTP, leading implementations to refine their behaviour, often diverging from the specification.
- o Likewise, changes such as HTTP/2 [RFC7540] might change the underlying assumptions that these requirements were based upon.
- o Emerging lower-layer developments such as TCP Fast Open [RFC7413], TLS/1.3 [I-D.ietf-tls-tls13] and QUIC [I-D.ietf-quic-transport] introduce the possibility of replayed requests in the beginning of a connection, thanks to Zero Round Trip (0RT) modes. In some ways, these are similar to retries - but not completely.
- o Applications sometimes want requests to be retried by infrastructure, but can't easily express them in a non-idempotent request (such as GET).

This draft gives some background in Section 2, discusses aspects of these issues in Section 3, suggesting possible areas of work in Section 4, and cataloguing current implementation behaviours in Appendix A.

### 1.1. Notational Conventions

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].

## 2. Background

### 2.1. Retries and Replays: A Taxonomy of Repetition

In HTTP, there are three similar but separate phenomena that deserve consideration for the purposes of this document:

1. \*User Retries\* happen when a user initiates an action that results in a duplicate HTTP request message being emitted. For example, a user retry might occur when a "reload" button is

pressed, a URL is typed in again, "return" is pressed in the URL bar again, or a navigation link or form button is pressed twice while still on screen.

2. \*Automatic Retries\* happen when an HTTP client implementation resends a previous request message without user intervention or initiation. This might happen when a GET request fails to return a complete response, or when a connection drops before the request is sent. Note that automatic retries can (and are) performed both by user agents and intermediary clients.
3. \*Replays\* happen when the underlying transport units (e.g., TCP packets, QUIC frames) containing a HTTP request message are re-sent on the network \*and\* appear to be separate requests to the downstream server, either automatically as part of transport protocol operation, or by an attacker. The upstream HTTP client might not have any indication that a replay has occurred.

Note that retries initiated by code shipped to the client by the server (e.g., in JavaScript) occupy a grey area here. Because they are not initiated by the generic HTTP client implementation itself, we will consider them user retries for the time being.

Also, this document doesn't include transport layer loss recovery (e.g., TCP retransmission). This is distinguished from replays because the transport automatically suppresses duplicates.

## 2.2. What the Spec Says: Automatic Retries

[RFC7230], Section 6.3.1 allows HTTP requests to be retried in certain circumstances:

When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of requests if all of those requests have idempotent methods (Section 4.2.2 of [RFC7231]). A proxy MUST NOT automatically retry non-idempotent requests.

A user agent MUST NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied. For example, a user agent that knows (through design or configuration) that a POST request to a given resource is safe can repeat that request automatically. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a

failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.

A client SHOULD NOT automatically retry a failed automatic retry.

Note that the complete list of idempotent methods is maintained in the IANA HTTP Method Registry [4].

## 2.3. What the Specs Say: Replay

### 2.3.1. TCP Fast Open

[RFC7413], Section 6.3.1 addresses HTTP Request Replay with TCP Fast Open:

While TFO is motivated by Web applications, the browser should not use TFO to send requests in SYNs if those requests cannot tolerate replays. One example is POST requests without application-layer transaction protection (e.g., a unique identifier in the request header).

On the other hand, TFO is particularly useful for GET requests. GET request replay could happen across striped TCP connections: after a server receives an HTTP request but before the ACKs of the requests reach the browser, the browser may time out and retry the same request on another (possibly new) TCP connection. This differs from a TFO replay only in that the replay is initiated by the browser, not by the TCP stack.

The same specification addresses HTTP over TLS in Section 6.3.2:

For Transport Layer Security (TLS) over TCP, it is safe and useful to include a TLS client\_hello in the SYN packet to save one RTT in the TLS handshake. There is no concern about violating idempotency. In particular, it can be used alone with the speculative connection above.

### 2.3.2. TLS 1.3

[I-D.ietf-tls-tls13], Section 2.3 explains the properties of Zero-RTT Data in TLS 1.3:

**IMPORTANT NOTE:** The security properties for 0-RTT data (regardless of the cipher suite) are weaker than those for other kinds of TLS data. Specifically:



1. This data is not forward secret, because it is encrypted solely with the PSK.
2. There are no guarantees of non-replay between connections. Unless the server takes special measures outside those provided by TLS, the server has no guarantee that the same 0-RTT data was not transmitted on multiple 0-RTT connections (See Section 4.2.6.2 for more details). This is especially relevant if the data is authenticated either with TLS client authentication or inside the application layer protocol. However, 0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.)

Section 4.2.6 defines a mechanism to limit the exposure to replay.

### 2.3.3. QUIC

[I-D.ietf-quic-tls] Section 7.2 says this about the risks of replay during the 0RTT handshake:

If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.

A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.

A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.

A server MUST NOT use 0-RTT keys to protect packets.

## 3. Discussion

### 3.1. Automatic Retries In Practice

In practice, it has been observed (see Appendix A) that some client implementations (both user agent and intermediary) do automatically retry requests. However, they do not do so consistently, and

arguably not in the spirit of the specification, unless this vague catch-all:

some means to detect that the original request was never applied is interpreted very broadly.

On the server side, it has been widely observed that content on the Web doesn't always honour HTTP idempotency semantics, with many GET requests incurring side effects, and with some sites even requiring browsers to retry POST requests in order to properly interoperate.

Despite this situation, the Web seems to work reasonably well to date (with notable exceptions [5]).

The status quo, therefore, is that no Web application can read HTTP's retry requirements as a guarantee that any given request won't be retried, even for methods that are not idempotent. As a result, applications that care about avoiding duplicate requests need to build a way to detect not only user retries but also automatic retries into the application "above" HTTP itself.

### 3.2. Replays Are Different

TCP Fast Open [RFC7413], TLS/1.3 [I-D.ietf-tls-tls13] and QUIC [I-D.ietf-quic-transport] all have mechanisms to carry application data on the first packet sent by a client, to avoid the latency of connection setup.

The request(s) in this first packet might be replayed, either because the first packet (now carrying a HTTP request) is thought to be lost and retransmitted, or because an attacker observes the packet and sends a duplicate at some point in the future.

At first glance, it seems as if the idempotency semantics of HTTP request methods could be used to determine what requests are suitable for inclusion in the first packet of various ORTT mechanisms being discussed (as suggested by TCP Fast Open). For example, we could disallow POST (and other non-idempotent methods) in ORTT data.

Upon reflection, though, the observations above lead us to believe that since any request might be retried (automatically or by users), applications will still need to have a means of detecting duplicate requests, thereby preventing side effects from replays as well as retries. Thus, any HTTP request can be included in the first packet of a ORTT, despite the risk of replay.

Two types of attack specific to replayed HTTP requests need to be taken into account, however:

1. A replay is a potential Denial of Service vector. An attacker that can replay a request many times can probe for weaknesses in retry protections, and can bring a server that needs to do any substantial processing down.
2. An attacker might use a replayed request to leak information about the response over time. If they can observe the encrypted payload on the wire, they can infer the size of the response (e.g., it might get bigger if the user's bank account has more in it).

The first attack cannot be mitigated by HTTP; the ORT mechanism itself needs some transport-layer means of scoping the usability of the first packet on a connection so that it cannot be reused broadly. For example, this might be by time, or by network location.

The second attack is more difficult to mitigate; scoping the usability of the first packet helps, but does not completely prevent the attack. If the replayed request is state-changing, the application's retry detection should kick in and prevent information leakage (since the response will likely contain an error, instead of the desired information).

If it is not (e.g., a GET), the information being targeted is vulnerable as long as both the first packet and the credentials in the request (if any) are valid.

#### 4. Possible Areas of Work

##### 4.1. Updating HTTP's Requirements for Retries

The currently language in [RFC7230] about retries is vague about the conditions under which a request can be retried, leading to significant variance in implementation behaviour. For example, it's been observed that many automated clients fail under circumstances when browsers succeed, because they do not retry in the same way.

As a result, more carefully specifying the conditions under which a request can be retried would be helpful. Such work would need to take into account varying conditions, such as:

- o Connection closes
- o TCP RST

- o Connection timeouts
- o Whether or not any part of the response has been received
- o Whether or not it is the first request on the connection
- o Variance due to use of HTTP/2, TLS/1.3, TCP Fast Open and QUIC.

Furthermore, readers might mistake the language in RFC7230 as guaranteeing that some requests (e.g., POST) are never automatically retried; this should be clarified.

#### 4.2. Protocol Extensions

A number of mechanisms have been mooted at various times, e.g.:

- o Adding a header to automatically retried requests, to aid de-duplication by servers
- o Defining a request header to be added by intermediaries when they have received a request in a way that could have been replayed
- o Defining a status code to allow servers to indicate that the request needs to be sent in a way that can't be replayed

#### 4.3. Feedback to Transport ORTT Efforts

If the observations above hold, we should disabuse any notion that HTTP method idempotency is a useful way to avoid problems with replay attacks. Instead, we should encourage development of mechanisms to mitigate the aspects of replay that are different than retries (e.g., potential for DOS attacks).

#### 5. Security Considerations

Yep.

#### 6. Acknowledgements

Thanks to Brad Fitzpatrick, Leif Hedstrom, Subodh Iyengar, Amos Jeffries, Patrick McManus, Matt Menke, Miroslav Ponec, Daniel Stenberg and Martin Thomson for their input and feedback.

Thanks also to the participants in the 2016 HTTP Workshop for their lively discussion of this topic.

## 7. References

### 7.1. Normative References

- [I-D.ietf-quic-tls]  
Thomson, M. and (. (Unknown), "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<http://www.rfc-editor.org/info/rfc7230>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<http://www.rfc-editor.org/info/rfc7231>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.

### 7.2. Informative References

- [I-D.ietf-quic-transport]  
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.
- [I-D.ietf-tls-tls13]  
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<http://www.rfc-editor.org/info/rfc7540>>.

### 7.3. URIs

- [1] <https://www.iana.org/assignments/http-methods/http-methods.xhtml>
- [2] [https://signalvnoise.com/archives2/google\\_web\\_accelerator\\_hey\\_not\\_so\\_fast\\_an\\_alert\\_for\\_web\\_app\\_designers.php](https://signalvnoise.com/archives2/google_web_accelerator_hey_not_so_fast_an_alert_for_web_app_designers.php)
- [3] <http://bazaar.launchpad.net/~squid/squid/trunk/view/head:/src/FwdState.cc#L594>
- [4] <https://git-wip-us.apache.org/repos/asf?p=trafficserver.git;a=blob;f=proxy/http/HttpTransact.cc;h=8a1f5364d47654b118296a07a2a95284f119d84b;hb=HEAD#l6408>
- [5] <https://git-wip-us.apache.org/repos/asf?p=trafficserver.git;a=blob;f=proxy/http/HttpTransact.cc;hb=48d7b25ba8a8229b0471d37cdaa6ef24cc634bb0#l3634>
- [6] <http://mxr.mozilla.org/mozilla-release/source/network/protocol/http/nsHttpTransaction.cpp#938>
- [7] <http://mxr.mozilla.org/mozilla-release/source/network/protocol/http/nsHttpRequestHead.cpp#67>
- [8] <https://www.fxsitecompat.com/en-CA/docs/2016/post-request-fails-on-certain-sites-showing-connection-reset-page/>
- [9] [https://chromium.googlesource.com/chromium/src.git/+/master/net/http/http\\_network\\_transaction.cc#l657](https://chromium.googlesource.com/chromium/src.git/+/master/net/http/http_network_transaction.cc#l657)
- [10] <https://github.com/curl/curl/blob/master/lib/transfer.c#L1892>

## Appendix A. When Clients Retry

In implementations, clients have been observed to retry requests in a number of circumstances.

Note: This section is intended to inform the discussion, not to be published as a standard. If you have relevant information about these or other implementations (open or closed), please get in touch.\_

### A.1. Squid

Squid is a caching proxy server that retries requests that it considers safe *\*or\** idempotent, as long as there is not a request body:

```
/// Whether we may try sending this request again after a failure.
bool
FwdState::checkRetriable()
{
    // Optimize: A compliant proxy may retry PUTs, but Squid lacks the [rather
    // complicated] code required to protect the PUT request body from being
    // nibbled during the first try. Thus, Squid cannot retry some PUTs today.
    if (request->body_pipe != NULL)
        return false;

    // RFC2616 9.1 Safe and Idempotent Methods
    return (request->method.isHttpSafe() || request->method.isIdempotent());
}
```

(source [6])

Currently, it considers GET, HEAD, OPTIONS, REPORT, PROPFIND, SEARCH and PRI to be safe, and GET, HEAD, PUT, DELETE, OPTIONS, TRACE, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, UNLOCK, and PRI to be idempotent.

#### A.2. Traffic Server

Apache Traffic Server, a caching proxy server, ties retry-ability to whether the request required a "tunnel" - i.e., forwarding the request body to the next server. This is indicated by "request\_body\_start", which is set when a POST tunnel is used.

```

// bool HttpTransact::is_request_retryable
//
//   If we started a POST/PUT tunnel then we can
//   not retry failed requests
//
bool
HttpTransact::is_request_retryable(State *s)
{
    if (s->hdr_info.request_body_start == true) {
        return false;
    }

    if (s->state_machine->plugin_tunnel_type != HTTP_NO_PLUGIN_TUNNEL) {
        // API can override
        if (s->state_machine->plugin_tunnel_type == HTTP_PLUGIN_AS_SERVER &&
            s->api_info.retry_intercept_failures == true) {
            // This used to be an == comparison, which made no sense. Changed
            // to be an assignment, hoping the state is correct.
            s->state_machine->plugin_tunnel_type = HTTP_NO_PLUGIN_TUNNEL;
        } else {
            return false;
        }
    }

    return true;
}

```

(source [7])

When connected to an origin server, Traffic Server attempts to retry under a number of failure conditions:

```

////////////////////////////////////
// Name      : handle_response_from_server
// Description: response is from the origin server
//
// Details   :
//
//   response from the origin server. one of three things can happen now.
//   if the response is bad, then we can either retry (by first downgrading
//   the request, maybe making it non-keepalive, etc.), or we can give up.
//   the latter case is handled by handle_server_connection_not_open and
//   sends an error response back to the client. if the response is good
//   handle_forward_server_connection_open is called.
//
//
// Possible Next States From Here:
//

```



```

////////////////////////////////////
void
HttpTransact::handle_response_from_server(State *s)
{
    [...]

    switch (s->current.state) {
    case CONNECTION_ALIVE:
        DebugTxn("http_trans", "[hrfs] connection alive");
        SET_VIA_STRING(VIA_DETAIL_SERVER_CONNECT, VIA_DETAIL_SERVER_SUCCESS);
        s->current.server->clear_connect_fail();
        handle_forward_server_connection_open(s);
        break;

    [...]

    case OPEN_RAW_ERROR:
        /* fall through */
    case CONNECTION_ERROR:
        /* fall through */
    case STATE_UNDEFINED:
        /* fall through */
    case INACTIVE_TIMEOUT:
        // Set to generic I/O error if not already set specifically.
        if (!s->current.server->had_connect_fail())
            s->current.server->set_connect_fail(EIO);

        if (is_server_negative_cached(s)) {
            max_connect_retries = s->txn_conf->connect_attempts_max_retries_dead_serve
r;
        } else {
            // server not yet negative cached - use default number of retries
            max_connect_retries = s->txn_conf->connect_attempts_max_retries;
        }
        if (s->pCongestionEntry != NULL)
            max_connect_retries = s->pCongestionEntry->connect_retries();

        if (is_request_retryable(s) && s->current.attempts < max_connect_retries) {
            (source [8])

```

### A.3. Firefox

Firefox is a Web browser that retries under the following conditions:

```
// if the connection was reset or closed before we wrote any part of the
// request or if we wrote the request but didn't receive any part of the
// response and the connection was being reused, then we can (and really
// should) assume that we wrote to a stale connection and we must therefore
// repeat the request over a new connection.
//
// We have decided to retry not only in case of the reused connections, but
// all safe methods(bug 1236277).
//
// NOTE: the conditions under which we will automatically retry the HTTP
// request have to be carefully selected to avoid duplication of the
// request from the point-of-view of the server. such duplication could
// have dire consequences including repeated purchases, etc.
//
// NOTE: because of the way SSL proxy CONNECT is implemented, it is
// possible that the transaction may have received data without having
// sent any data. for this reason, mSendData == FALSE does not imply
// mReceivedData == FALSE. (see bug 203057 for more info.)
//
[...]
```

```
if (!mReceivedData &&
    ((mRequestHead && mRequestHead->IsSafeMethod()) ||
     !reallySentData || connReused)) {
    // if restarting fails, then we must proceed to close the pipe,
    // which will notify the channel that the transaction failed.

    (source [9])

    ... and it considers GET, HEAD, OPTIONS, TRACE, PROPFIND, REPORT, and
    SEARCH to be safe:
```

```
bool
nsHttpRequestHead::IsSafeMethod() const
{
    // This code will need to be extended for new safe methods, otherwise
    // they'll default to "not safe".
    if (IsGet() || IsHead() || IsOptions() || IsTrace()) {
        return true;
    }

    if (mParsedMethod != kMethod_Custom) {
        return false;
    }

    return (!strcmp(mMethod.get(), "PROPFIND") ||
            !strcmp(mMethod.get(), "REPORT") ||
            !strcmp(mMethod.get(), "SEARCH"));
}

(source [10])
```

Note that "connReused" is tested; if a connection has been used before, Firefox will retry any request, safe or not. A recent change attempted to remove this behaviour, but it caused compatibility problems [11], and is being backed out.

#### A.4. Chromium

Chromium is a Web browser that appears to retry any request when a connection is broken, as long as it's successfully used the connection before, and hasn't received any response headers yet:

```
bool HttpNetworkTransaction::ShouldResendRequest() const {
    bool connection_is_proven = stream_>IsConnectionReused();
    bool has_received_headers = GetResponseHeaders() != NULL;

    // NOTE: we resend a request only if we reused a keep-alive connection.
    // This automatically prevents an infinite resend loop because we'll run
    // out of the cached keep-alive connections eventually.
    if (connection_is_proven && !has_received_headers)
        return true;
    return false;
}

(source [12])
```

## A.5. Curl

Curl is both a command-line client and widely-used library for HTTP. Like Chromium, it will retry a request if the response hasn't started.

```
CURLcode Curl_retry_request(struct connectdata *conn,
                           char **url)
{
    struct Curl_easy *data = conn->data;

    *url = NULL;

    /* if we're talking upload, we can't do the checks below, unless the protocol
       is HTTP as when uploading over HTTP we will still get a response */
    if(data->set.upload &&
        !(conn->handler->protocol&(PROTO_FAMILY_HTTP|CURLPROTO_RTSP)))
        return CURLE_OK;

    if((data->req.bytecount + data->req.headerbytecount == 0) &&
        conn->bits.reuse &&
        (data->set.rtspreq != RTSPREQ_RECEIVE)) {
        /* We didn't get a single byte when we attempted to re-use a
           connection. This might happen if the connection was left alive when we
           were done using it before, but that was closed when we wanted to use it
           again. Bad luck. Retry the same request on a fresh connect! */
        infof(conn->data, "Connection died, retrying a fresh connect\n");
        *url = strdup(conn->data->change.url);
        if(!*url)
            return CURLE_OUT_OF_MEMORY;

        connclose(conn, "retry"); /* close this connection */
        conn->bits.retry = TRUE; /* mark this as a connection we're about
                                to retry. Marking it this way should
                                prevent i.e HTTP transfers to return
                                error just because nothing has been
                                transferred! */

        if(conn->handler->protocol&PROTO_FAMILY_HTTP) {
            struct HTTP *http = data->req.protop;
            if(http->writebytecount)
                return Curl_readrewind(conn);
        }
    }
    return CURLE_OK;
}
```

(source [13])

Author's Address

Mark Nottingham

Email: [mnot@mnot.net](mailto:mnot@mnot.net)

URI: <https://www.mnot.net/>