Abstract

This document specifies an "HTTP" resource record type for the DNS to facilitate the lookup of the server hostname of HTTP(s) URIs. It is intended to replace the use of CNAME records for this purpose, and in the process provides a solution for the inability of the DNS to allow a CNAME to be placed at the apex of a domain name.

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 May 8, 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.
1. Introduction

It is very common for HTTP(s) URIs to contain a domain name that is not the same as the hostname of the actual server that hosts the content.

This is typically achieved via a CNAME record where the owner name of that record (the "Alias") is the domain name from the URI and the Canonical name field in its RDATA corresponds with the target hostname (although it should be noted that this strictly a violation of the original design semantics of the CNAME record).

It is also impossible to store a CNAME at the apex of a domain name, which causes significant difficulties if you wish to redirect your domain name without a "www" prefix to a content delivery network (CDN). The only portable solution at the moment is to determine the IP address records of the content host and insert them directly at the apex of the zone, but this is brittle, and prevents the correct operation of typical CDN features.

While there have been previous attempts to promote the use of the SRV record instead of CNAME records, there have been concerns raised about the performance impact of the additional DNS lookup an SRV record would typically require.

To achieve equivalent end-user performance as existing CNAME-based solutions, this document permits recursive resolvers to pre-emptively look up the target of an HTTP Record and return the corresponding
records to the client. While this feature is not mandatory it is
hoped that support would over time become near ubiquitous.

Also, the presence of the Port field in an SRV record is incompatible
with the "Same Origin" security policy enforced by web browsers and
in practice the load-balancing / fallback capabilities of the SRV
record are not widely used either, and non-DNS based solutions for
this are already widely deployed for HTTP traffic.

This document therefore specifies a minimal "HTTP" resource record
type for the DNS to facilitate the redirection from the domain name
portion of an HTTP(s) URI to the server hostname and thence to A or
AAAA records. It is specifically intended to replace the use of
CNAME records for this purpose, and in the process provides a
solution for the inability of the DNS to allow a CNAME to be placed
at the apex of a domain name.

2. 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.

3. Description

The owner name of an HTTP RR is the domain name portion of an HTTP(s)
URI.

The use of underscore label prefixes (e.g. _http._tcp) was
considered, but rejected since it prohibits the use of wildcard
records which us a valuable technique for offering per-customer
domain prefixes without requiring that every prefix be individually
provisioned.

3.1. Wire Format

The RDATA of an HTTP RR is a domain name in uncompressed wire format.

3.2. Presentation Format

The RDATA of an HTTP RR is presented as a domain name in standard
master file format.
3.3. Server Operation

Recursive resolvers MAY on receiving a request for an HTTP record look up the A and AAAA records for the target (either from cache, or via new iterative queries) and include the results in the Additional Section of the response.

If the recursive resolver is performing DNSSEC resolution but is unable to validate the A or AAAA responses it MUST NOT include them in the response unless the client has specified the +CD (checking disabled) flag.

Where EDNS Client Subnet [RFC7871] is configured on the resolver those A and AAAA lookups MUST be performed as if the client had made those queries directly to the resolver.

3.4. Client Operation

HTTP clients supporting this specification MUST issue parallel DNS requests for the A, AAAA and HTTP records for the domain portion of an http: or https: URI.

If an HTTP record is returned, the client MUST either use the A and AAAA records contained in the Additional Section of the response, or issue further parallel requests for the A and AAAA records corresponding to the domain name in the RDATA of the HTTP record and then use those IP addresses to access the URI.

If the original A and AAAA lookups return IP addresses these MUST only be used if no HTTP record is returned.

<< the above needs more text around timing, happy eyeballs, etc. >>

4. Security Considerations

TBD

5. Implementation status

<< RFC Editor Note: Please remove this entire section prior to publication as an RFC. >>

6. Privacy Considerations

TBD (if any)
7. IANA Considerations

<< a copy of the RFC 6895 IANA RR TYPE application template will appear here >>

8. Acknowledgements

9. References

9.1. Normative References


9.2. Informative References


Author’s Address

Ray Bellis
Internet Systems Consortium, Inc.
950 Charter Street
Redwood City CA 94063
USA

Phone: +1 650 423 1200
Email: ray@isc.org
Abstract

HTTP is often used as a substrate for other application protocols (a.k.a. HTTP-based APIs). This document specifies best practices for such protocols’ use of HTTP when they are defined for diverse implementation and broad deployment (e.g., in standards efforts).

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (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/bcp56bis [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 May 13, 2019.
Table of Contents

1.  Introduction .................................................. 3
   1.1.  Notational Conventions .................................. 4
2.  Is HTTP Being Used? ........................................ 4
3.  What’s Important About HTTP ................................. 5
   3.1.  Generic Semantics ........................................ 5
   3.2.  Links ...................................................... 6
   3.3.  Rich Functionality ....................................... 7
4.  Best Practices for Using HTTP ............................... 8
   4.1.  Specifying the Use of HTTP ............................. 8
   4.2.  Defining HTTP Resources ................................. 9
   4.3.  Specifying Client Behaviours ........................... 9
   4.4.  HTTP URLs ................................................. 10
      4.4.1.  Initial URL Discovery ............................... 11
      4.4.2.  URL Schemes ......................................... 11
      4.4.3.  Transport Ports ...................................... 12
   4.5.  HTTP Methods .............................................. 13
      4.5.1.  GET .................................................. 13
      4.5.2.  OPTIONS .............................................. 14
   4.6.  HTTP Status Codes ....................................... 15
      4.6.1.  Redirection ........................................... 16
   4.7.  HTTP Header Fields ...................................... 17
   4.8.  Defining Message Payloads ............................... 18
   4.9.  HTTP Caching .............................................. 18
   4.10. Application State ........................................ 20
   4.11. Client Authentication ................................... 21
   4.12. Co-Existing with Web Browsing .......................... 21
   4.13. Application Boundaries .................................. 23
   4.15. Versioning and Evolution ................................ 24
5.  IANA Considerations ........................................ 25
6.  Security Considerations .................................... 25
6.1. Privacy Considerations

7. References

7.1. Normative References

7.2. Informative References

7.3. URIs

Appendix A. Changes from RFC 3205

Author's Address

1. Introduction

HTTP [I-D.ietf-httpbis-semantics] is often used as a substrate for applications other than Web browsing; this is sometimes referred to as creating "HTTP-based APIs", or just "HTTP APIs". This is done for a variety of reasons, including:

- familiarity by implementers, specifiers, administrators, developers and users,
- availability of a variety of client, server and proxy implementations,
- ease of use,
- availability of Web browsers,
- reuse of existing mechanisms like authentication and encryption,
- presence of HTTP servers and clients in target deployments, and
- its ability to traverse firewalls.

These protocols are often ad hoc; they are intended for only deployment by one or a few servers, and consumption by a limited set of clients. Perhaps because of the factors cited above, a body of practices and tools has arisen around defining HTTP-based APIs that favours these conditions.

However, when such an application has multiple, separate implementations of the server component, is deployed on multiple uncoordinated servers, and is consumed by diverse clients – as is often the case for standards efforts to define new HTTP APIs – tools and practices intended for limited deployment can become unsuitable.

This is largely because implementations (both client and server) will implement and evolve at different paces. As a result, such an HTTP-based API will need to more carefully consider how extensibility of the service will be handled and how different deployment requirements will be accommodated.
More generally, application protocols using HTTP face a number of design decisions, including:

- Should it define a new URL scheme? Use new ports?
- Should it use standard HTTP methods and status codes, or define new ones?
- How can the maximum value be extracted from the use of HTTP?
- How does it coexist with other uses of HTTP - especially Web browsing?
- How can interoperability problems and "protocol dead ends" be avoided?

This document contains best current practices regarding the use of HTTP by applications other than Web browsing. Section 2 defines what applications it applies to; Section 3 surveys the properties of HTTP that are important to preserve, and Section 4 conveys best practices for those applications that do use HTTP.

It is written primarily to guide IETF efforts to define application protocols using HTTP for deployment on the Internet, but might be applicable in other situations. Note that the requirements herein do not necessarily apply to the development of generic HTTP extensions.

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. Is HTTP Being Used?

Different applications have different goals when using HTTP. The requirements in this document apply when any of the following conditions are true:

- The transport port in use is 80 or 443,
- The URL scheme "http" or "https" is used,
- The ALPN protocol ID [RFC7301] generically identifies HTTP (e.g., "http/1.1", "h2", "h2c"), or
The IANA registries defined for HTTP are updated or modified.

When an application is using HTTP, all of the requirements of the HTTP protocol suite are in force (including but not limited to
[I-D.ietf-httpbis-semantics], [I-D.ietf-httpbis-cache],
[I-D.ietf-httpbis-messaging], and [RFC7540]).

An application might not use HTTP according to this definition and
still rely upon the HTTP specifications in some manner. For example,
an application might wish to avoid re-specifying parts of the message
format, but change others; or, it might want to use a different set
of methods.

Such applications are referred to as "protocols based upon HTTP" in
this document. These have more freedom to modify protocol
operations, but are also likely to lose at least a portion of the
benefits outlined above, as most HTTP implementations won’t be easily
adaptable to these changes, and as the protocol diverges from HTTP,
the benefit of mindshare will be lost.

Protocols that are based upon HTTP MUST NOT reuse HTTP’s URL schemes,
transport ports, ALPN protocol IDs or IANA registries; rather, they
are encouraged to establish their own.

3. What’s Important About HTTP

Applications using HTTP are defined and deployed in many ways;
sometimes they are brought to the IETF for standardisation. What
might be workable for deployment in a limited fashion isn’t
appropriate for standardisation and the corresponding broader
deployment.

This section examines the facets of the protocol that are important
to preserve in these situations.

3.1. Generic Semantics

When writing a specification, it’s often tempting to specify exactly
how HTTP is to be implemented, supported and used.

However, this can easily lead to an unintended profile of HTTP’s
behaviour. For example, it’s common to see specifications with
language like this:

A 'POST' request MUST result in a '201 Created' response.

This forms an expectation in the client that the response will always
be "201 Created", when in fact there are a number of reasons why the
status code might differ in a real deployment. If the client does not anticipate this, the application’s deployment is brittle.

Much of the value of HTTP is in its generic semantics – that is, the protocol elements defined by HTTP are potentially applicable to every resource, not specific to a particular context. Application-specific semantics are expressed in the payload; mostly, in the body, but also in header fields.

This allows a HTTP message to be examined by generic software (e.g., HTTP servers, intermediaries, client implementations, and caches) and its handling to be correctly determined. It also allows people to leverage their knowledge of HTTP semantics without special-casing them for a particular application.

Therefore, applications that use HTTP MUST NOT re-define, refine or overlay the semantics of defined protocol elements. Instead, they should focus their specifications on protocol elements that are specific to that application; namely their HTTP resources.

See Section 4.2 for details.

3.2. Links

Another common practice is assuming that the HTTP server’s name space (or a portion thereof) is exclusively for the use of a single application. This effectively overlays special, application-specific semantics onto that space, precludes other applications from using it.

As explained in [RFC7320], such "squatting" on a part of the URL space by a standard usurps the server’s authority over its own resources, can cause deployment issues, and is therefore bad practice in standards.

Instead of statically defining URL components like paths, it is RECOMMENDED that applications using HTTP define links in payloads, to allow flexibility in deployment.

Using runtime links in this fashion has a number of other benefits – especially when an application is to have multiple implementations and/or deployments (as is often the case for those that are standardised).

For example, navigating with a link allows a request to be routed to a different server without the overhead of a redirection, thereby supporting deployment across machines well.
It also becomes possible to "mix and match" different applications on the same server, and offers a natural mechanism for extensibility, versioning and capability management, since the document containing the links can also contain information about their targets.

Using links also offers a form of cache invalidation that’s seen on the Web; when a resource’s state changes, the application can change its link to it so that a fresh copy is always fetched.

3.3. Rich Functionality

HTTP offers a number of features to applications, such as:

- Message framing
- Multiplexing (in HTTP/2)
- Integration with TLS
- Support for intermediaries (proxies, gateways, Content Delivery Networks)
- Client authentication
- Content negotiation for format, language, and other features
- Caching for server scalability, latency and bandwidth reduction, and reliability
- Granularity of access control (through use of a rich space of URLs)
- Partial content to selectively request part of a response
- The ability to interact with the application easily using a Web browser

Applications that use HTTP are encouraged to utilise the various features that the protocol offers, so that their users receive the maximum benefit from it, and to allow it to be deployed in a variety of situations. This document does not require specific features to be used, since the appropriate design tradeoffs are highly specific to a given situation. However, following the practices in Section 4 is a good starting point.
4. Best Practices for Using HTTP

This section contains best practices regarding the use of HTTP by applications, including practices for specific HTTP protocol elements.

4.1. Specifying the Use of HTTP

When specifying the use of HTTP, an application SHOULD use [I-D.ietf-httpbis-semantics] as the primary reference; it is not necessary to reference all of the specifications in the HTTP suite unless there are specific reasons to do so (e.g., a particular feature is called out).

Applications using HTTP SHOULD NOT specify a minimum version of HTTP to be used; because it is a hop-by-hop protocol, a HTTP connection can be handled by implementations that are not controlled by the application; for example, proxies, CDNs, firewalls and so on. Requiring a particular version of HTTP makes it difficult to use in these situations, and harms interoperability for little reason (since HTTP’s semantics are stable between protocol versions).

However, if an application’s deployment would benefit from the use of a particular version of HTTP (for example, HTTP/2’s multiplexing), this SHOULD be noted.

Applications using HTTP MUST NOT specify a maximum version, to preserve the protocol’s ability to evolve.

When specifying examples of protocol interactions, applications SHOULD document both the request and response messages, with full headers, preferably in HTTP/1.1 format. For example:

```
GET /thing HTTP/1.1
Host: example.com
Accept: application/things+json
User-Agent: Foo/1.0

HTTP/1.1 200 OK
Content-Type: application/things+json
Content-Length: 500
Server: Bar/2.2

[payload here]
```
4.2. Defining HTTP Resources

Applications that use HTTP should focus on defining the following application-specific protocol elements:

- Media types [RFC6838], often based upon a format convention such as JSON [RFC8259],
- HTTP header fields, as per Section 4.7, and
- The behaviour of resources, as identified by link relations [RFC8288].

By composing these protocol elements, an application can define a set of resources, identified by link relations, that implement specified behaviours, including:

- Retrieval of their state using GET, in one or more formats identified by media type;
- Resource creation or update using POST or PUT, with an appropriately identified request body format;
- Data processing using POST and identified request and response body format(s); and
- Resource deletion using DELETE.

For example, an application might specify:

Resources linked to with the "example-widget" link relation type are Widgets. The state of a Widget can be fetched in the "application/example-widget+json" format, and can be updated by PUT to the same link. Widget resources can be deleted.

The "Example-Count" response header field on Widget representations indicates how many Widgets are held by the sender.

The "application/example-widget+json" format is a JSON [RFC8259] format representing the state of a Widget. It contains links to related information in the link indicated by the Link header field value with the "example-other-info" link relation type.

4.3. Specifying Client Behaviours

Some behaviours (e.g., automatic redirect handling) and extensions (e.g., Cookies) are not required by HTTP, but nevertheless have become very common, possibly because they are supported by Web
browsers. If their use is not explicitly specified by applications using HTTP, there may be confusion and interoperability problems. This section recommends default handling for these mechanisms.

- Redirect handling - Applications need to specify how redirects are expected to be handled; see Section 4.6.1.

- Cookies - Applications using HTTP MUST explicitly reference the Cookie specification [I-D.ietf-httpbis-rfc6265bis] if they are required.

- Certificates - Applications using HTTP MUST specify that TLS certificates are to be checked according to [RFC2818] when HTTPS is used.

In general, applications using HTTP ought to align their usage as closely as possible with Web browsers, to avoid interoperability issues when they are used. See Section 4.12.

If an application using HTTP has browser compatibility as a goal, client interaction ought to be defined in terms of [FETCH], since that is the abstraction that browsers use for HTTP; it enforces many of these best practices.

Applications using HTTP MUST NOT require HTTP features that are usually negotiated to be supported by clients. For example, requiring that clients support responses with a certain content-coding ([I-D.ietf-httpbis-semantics], Section 6.2.2) instead of negotiating for it ([I-D.ietf-httpbis-semantics], Section 8.4.4) means that otherwise conformant clients cannot interoperate with the application. Applications MAY encourage the implementation of such features, though.

### 4.4. HTTP URLs

In HTTP, URLs are opaque identifiers under the control of the server. As outlined in [RFC7320], standards cannot usurp this space, since it might conflict with existing resources, and constrain implementation and deployment.

In other words, applications that use HTTP shouldn’t associate application semantics with specific URL paths on arbitrary servers. Doing so inappropriately conflates the identity of the resource (its URL) with the capabilities that resource supports, bringing about many of the same interoperability problems that [RFC4367] warns of.
For example, specifying that a "GET to the URL /foo retrieves a bar
document" is bad practice. Likewise, specifying "The widget API is
at the path /bar" violates [RFC7320].

Instead, applications are encouraged to ensure that URLs are
discovered at runtime, allowing HTTP-based services to describe their
own capabilities. One way to do this is to use typed links [RFC8288]
to convey the URIs that are in use, as well as the semantics of the
resources that they identify. See Section 4.2 for details.

4.4.1. Initial URL Discovery

Generally, a client will begin interacting with a given application
server by requesting an initial document that contains information
about that particular deployment, potentially including links to
other relevant resources.

Applications are encouraged to allow an arbitrary URL to be used as
that entry point. For example, rather than specifying "the initial
document is at "/foo/v1", they should allow a deployment to use any
URL as the entry point for the application.

In cases where doing so is impractical (e.g., it is not possible to
convey a whole URL, but only a hostname) applications can request a
well-known URL [I-D.nottingham-rfc5785bis] as an entry point.

4.4.2. URL Schemes

Applications that use HTTP will typically employ the "http" and/or
"https" URL schemes. "https" is RECOMMENDED to provide
authentication, integrity and confidentiality, as well as mitigate
pervasive monitoring attacks [RFC7258].

However, application-specific schemes can also be defined. When
defining an URL scheme for an application using HTTP, there are a
number of tradeoffs and caveats to keep in mind:

- Unmodified Web browsers will not support the new scheme. While it
  is possible to register new URL schemes with Web browsers (e.g.
  registerProtocolHandler() in [HTML5], as well as several
  proprietary approaches), support for these mechanisms is not
  shared by all browsers, and their capabilities vary.

- Existing non-browser clients, intermediaries, servers and
  associated software will not recognise the new scheme. For
  example, a client library might fail to dispatch the request; a
  cache might refuse to store the response, and a proxy might fail
  to forward the request.
Because URLs occur in HTTP artefacts commonly, often being generated automatically (e.g., in the "Location" response header), it can be difficult to assure that the new scheme is used consistently.

The resources identified by the new scheme will still be available using "http" and/or "https" URLs. Those URLs can "leak" into use, which can present security and operability issues. For example, using a new scheme to assure that requests don’t get sent to a "normal" Web site is likely to fail.

Features that rely upon the URL’s origin [RFC6454], such as the Web’s same-origin policy, will be impacted by a change of scheme.

HTTP-specific features such as cookies [I-D.ietf-httpbis-rfc6265bis], authentication [I-D.ietf-httpbis-semantics], caching [I-D.ietf-httpbis-cache], HSTS [RFC6797], and CORS [FETCH] might or might not work correctly, depending on how they are defined and implemented. Generally, they are designed and implemented with an assumption that the URL will always be "http" or "https".

Web features that require a secure context [SECCTXT] will likely treat a new scheme as insecure.

See [RFC7595] for more information about minting new URL schemes.

4.4.3. Transport Ports

Applications can use the applicable default port (80 for HTTP, 443 for HTTPS), or they can be deployed upon other ports. This decision can be made at deployment time, or might be encouraged by the application’s specification (e.g., by registering a port for that application).

If a non-default port is used, it needs to be reflected in the authority of all URLs for that resource; the only mechanism for changing a default port is changing the scheme (see Section 4.4.2).

Using a port other than the default has privacy implications (i.e., the protocol can now be distinguished from other traffic), as well as operability concerns (as some networks might block or otherwise interfere with it). Privacy implications should be documented in Security Considerations.

See [RFC7605] for further guidance.
4.5. HTTP Methods

Applications that use HTTP MUST confine themselves to using registered HTTP methods such as GET, POST, PUT, DELETE, and PATCH.

New HTTP methods are rare; they are required to be registered in the HTTP Method Registry with IETF Review (see [I-D.ietf-httpbis-semantics]), and are also required to be generic. That means that they need to be potentially applicable to all resources, not just those of one application.

While historically some applications (e.g., [RFC4791]) have defined non-generic methods, [I-D.ietf-httpbis-semantics] now forbids this.

When authors believe that a new method is required, they are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application’s specification.

4.5.1. GET

GET is one of the most common and useful HTTP methods; its retrieval semantics allow caching, side-effect free linking and underlies many of the benefits of using HTTP.

A common use of GET is to perform queries, often using the query component of the URL; this is a familiar pattern from Web browsing, and the results can be cached, improving efficiency of an often expensive process.

In some cases, however, GET might be unwieldy for expressing queries, because of the limited syntax of the URL; in particular, if binary data forms part of the query terms, it needs to be encoded to conform to URL syntax.

While this is not an issue for short queries, it can become one for larger query terms, or ones which need to sustain a high rate of requests. Additionally, some HTTP implementations limit the size of URLs they support - although modern HTTP software has much more generous limits than previously (typically, considerably more than 8000 octets, as required by [I-D.ietf-httpbis-semantics]).

In these cases, an application using HTTP might consider using POST to express queries in the request body; doing so avoids encoding overhead and URL length limits in implementations. However, in doing so it should be noted that the benefits of GET such as caching and linking to query results are lost. Therefore, applications using
HTTP that feel a need to allow POST queries ought consider allowing both methods.

Applications SHOULD NOT define GET requests to have side effects, since implementations can and do retry HTTP GET requests that fail.

Finally, note that while HTTP allows GET requests to have a body syntactically, this is done only to allow parsers to be generic; as per [I-D.ietf-httpbis-semantics], Section 7.3.1, a body on a GET has no meaning, and will be either ignored or rejected by generic HTTP software.

4.5.2. OPTIONS

The OPTIONS method was defined for metadata retrieval, and is used both by WebDAV [RFC4918] and CORS [FETCH]. Because HTTP-based APIs often need to retrieve metadata about resources, it is often considered for their use.

However, OPTIONS does have significant limitations:

- It isn’t possible to link to the metadata with a simple URL, because OPTIONS is not the default GET method.
- OPTIONS responses are not cacheable, because HTTP caches operate on representations of the resource (i.e., GET and HEAD). If OPTIONS responses are cached separately, their interaction with HTTP cache expiry, secondary keys and other mechanisms needs to be considered.
- OPTIONS is "chatty" - always separating metadata out into a separate request increases the number of requests needed to interact with the application.
- Implementation support for OPTIONS is not universal; some servers do not expose the ability to respond to OPTIONS requests without significant effort.

Instead of OPTIONS, one of these alternative approaches might be more appropriate:

- For server-wide metadata, create a well-known URI [I-D.nottingham-rfc5785bis], or using an already existing one if it’s appropriate (e.g., HostMeta [RFC6415]).
- For metadata about a specific resource, create a separate resource and link to it using a Link response header or a link serialised into the representation’s body. See [RFC8288]. Note that the
Link header is available on HEAD responses, which is useful if the client wants to discover a resource’s capabilities before they interact with it.

4.6. HTTP Status Codes

The primary function of a HTTP status code is to convey semantics for the benefit of generic HTTP software, not to convey application-specific semantics.

Status codes are often generated or overwritten by intermediaries, as well as server and client implementations. This can happen, for example, when network errors are encountered, a captive portal is present, when an implementation is overloaded, or it thinks it is under attack. As a result, the status code that a server-side application generates and the one that the client software receives often differ.

This means that status codes are not a reliable way to carry application-specific signals. Specifying that a particular status code has a specific meaning in the context of an application can have unintended side effects; if that status code is generated by a generic HTTP component can lead clients to believe that the application is in a state that wasn’t intended.

Instead, applications using HTTP should specify the implications of general classes of responses (e.g., "successful response" for 2xx; "client error" for 4xx and "server error" for 5xx), conveying any application-specific information in the message body and/or HTTP header fields, not the status code. [RFC7807] provides one way for applications using HTTP to do so for error conditions.

There are limited exceptions to this; for example, applications might use 201 (Created) or 404 (Not Found) to convey application semantics that are compatible with the generic HTTP semantics of those status codes. In general, though, applications should resist the temptation to map their semantics into fine-grained status codes.

Because the set of registered HTTP status codes can expand, applications using HTTP should explicitly point out that clients ought to be able to handle all applicable status codes gracefully (i.e., falling back to the generic "n00" semantics of a given status code; e.g., "499" can be safely handled as "400" by clients that don’t recognise it). This is preferable to creating a "laundry list" of potential status codes, since such a list is never complete.

Applications using HTTP MUST NOT re-specify the semantics of HTTP status codes, even if it is only by copying their definition. They
MUST NOT require specific reason phrases to be used; the reason phrase has no function in HTTP, is not guaranteed to be preserved by implementations, and the reason phrase is not carried at all in the HTTP/2 [RFC7540] message format.

Applications MUST only use registered HTTP status codes. As with methods, new HTTP status codes are rare, and required (by [I-D.ietf-httpbis-semantics]) to be registered with IETF Review. Similarly, HTTP status codes are generic; they are required (by [I-D.ietf-httpbis-semantics]) to be potentially applicable to all resources, not just to those of one application.

When authors believe that a new status code is required, they are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application’s specification.

4.6.1. Redirection

The 3xx series of status codes specified in [I-D.ietf-httpbis-semantics], Section 9.4 direct the user agent to another resource to satisfy the request. The most common of these are 301, 302, 307 and 308 ([RFC7538]), all of which use the Location response header field to indicate where the client should send the request to.

There are two ways that this group of status codes differ:

- Whether they are permanent or temporary. Permanent redirects can be used to update links stored in the client (e.g., bookmarks), whereas temporary ones can not. Note that this has no effect on HTTP caching; it is completely separate.

- Whether they allow the redirected request to change the request method from POST to GET. Web browsers generally do change POST to GET for 301 and 302; therefore, 308 and 307 were created to allow redirection without changing the method.

This table summarises their relationships:

<table>
<thead>
<tr>
<th></th>
<th>Permanent</th>
<th>Temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows changing the request method from POST to GET</td>
<td>301</td>
<td>302</td>
</tr>
<tr>
<td>Does not allow changing the request method</td>
<td>308</td>
<td>307</td>
</tr>
</tbody>
</table>
As noted in [I-D.ietf-httpbis-semantics], a user agent is allowed to automatically follow a 3xx redirect that has a Location response header field, even if they don’t understand the semantics of the specific status code. However, they aren’t required to do so; therefore, if an application using HTTP desires redirects to be automatically followed, it needs to explicitly specify the circumstances when this is required.

Applications using HTTP SHOULD specify that 301 and 302 responses change the subsequent request method from POST (but no other method) to GET, to be compatible with browsers.

Generally, when a redirected request is made, its header fields are copied from the original request’s. However, they can be modified by various mechanisms; e.g., sent Authorization ([I-D.ietf-httpbis-semantics]) and Cookie ([I-D.ietf-httpbis-rfc6265bis]) headers will change if the origin (and sometimes path) of the request changes. Applications using HTTP SHOULD specify if any request headers need to be modified or removed upon a redirect; however, this behaviour cannot be relied upon, since a generic client (like a browser) will be unaware of such requirements.

4.7. HTTP Header Fields

Applications MAY define new HTTP header fields. Typically, using HTTP header fields is appropriate in a few different situations:

- Their content is useful to intermediaries (who often wish to avoid parsing the body), and/or
- Their content is useful to generic HTTP software (e.g., clients, servers), and/or
- It is not possible to include their content in the message body (usually because a format does not allow it).

New header fields MUST be registered, as per [I-D.ietf-httpbis-semantics].

See [I-D.ietf-httpbis-semantics], Section 4.1.3 for guidelines to consider when minting new header fields. [I-D.ietf-httpbis-header-structure] provides a common structure for new header fields, and avoids many issues in their parsing and handling; it is RECOMMENDED that new header fields use it.

It is RECOMMENDED that header field names be short (even when HTTP/2 header compression is in effect, there is an overhead) but
appropriately specific. In particular, if a header field is specific to an application, an identifier for that application SHOULD form a prefix to the header field name, separated by a "-".

For example, if the "example" application needs to create three headers, they might be called "example-foo", "example-bar" and "example-baz". Note that the primary motivation here is to avoid consuming more generic header names, not to reserve a portion of the namespace for the application; see [RFC6648] for related considerations.

The semantics of existing HTTP header fields MUST NOT be re-defined without updating their registration or defining an extension to them (if allowed). For example, an application using HTTP cannot specify that the "Location" header has a special meaning in a certain context.

See Section 4.9 for the interaction between headers and HTTP caching; in particular, request headers that are used to "select" a response have impact there, and need to be carefully considered.

See Section 4.10 for considerations regarding header fields that carry application state (e.g., Cookie).

4.8. Defining Message Payloads

There are many potential formats for payloads; for example, JSON [RFC8259], XML [XML], and CBOR [RFC7049]. Best practices for their use are out of scope for this document.

Applications SHOULD register distinct media types for each format they define; this makes it possible to identify them unambiguously and negotiate for their use. See [RFC6838] for more information.

4.9. HTTP Caching

HTTP caching [I-D.ietf-httpbis-cache] is one of the primary benefits of using HTTP for applications; it provides scalability, reduces latency and improves reliability. Furthermore, HTTP caches are readily available in browsers and other clients, networks as forward and reverse proxies, Content Delivery Networks and as part of server software.

Assigning even a short freshness lifetime ([I-D.ietf-httpbis-cache], Section 4.2) - e.g., 5 seconds - allows a response to be reused to satisfy multiple clients, and/or a single client making the same request repeatedly. In general, if it is safe to reuse something, consider assigning a freshness lifetime; cache implementations take
active measures to remove content intelligently when they are out of space, so "it will fill up the cache" is not a valid concern.

The most common method for specifying freshness is the max-age response directive ([I-D.ietf-httpbis-cache], Section 5.2.2.8). The Expires header ([I-D.ietf-httpbis-cache], Section 5.3) can also be used, but it is not necessary to specify it; all modern cache implementations support Cache-Control, and specifying freshness as a delta is usually more convenient and always less error-prone.

Understand that stale responses (e.g., with "Cache-Control: max-age=0") can be reused when the cache is disconnected from the origin server; this can be useful for handling network issues. See [I-D.ietf-httpbis-cache], Section 4.2.4, and also [RFC5861] for additional controls over stale content.

Stale responses can be refreshed by assigning a validator, saving both transfer bandwidth and latency for large responses; see [I-D.ietf-httpbis-semantics].

If an application uses a request header field to change the response’s headers or body, authors should point out that this has implications for caching; in general, such resources need to either make their responses uncacheable (e.g., with the "no-store" cache-control directive defined in [I-D.ietf-httpbis-cache], Section 5.2.2.3) or send the Vary response header ([I-D.ietf-httpbis-semantics], Section 10.1.4) on all responses from that resource (including the "default" response).

For example, this response:

HTTP/1.1 200 OK
Content-Type: application/example+xml
Cache-Control: max-age=60
ETag: "sa0f8wf20fs0f"
Vary: Accept-Encoding

[content]

can be stored for 60 seconds by both private and shared caches, can be revalidated with If-None-Match, and varies on the Accept-Encoding request header field.

In some situations, responses without explicit cache directives (e.g., Cache-Control or Expires) will be stored and served using a heuristic freshness lifetime; see [I-D.ietf-httpbis-cache], Section 4.2.2. As the heuristic is not under control of the
application, it is generally preferable to set an explicit freshness lifetime.

If caching of a response is not desired, the appropriate response directive is "Cache-Control: no-store". This only need be sent in situations where the response might be cached; see [I-D.ietf-httpbis-cache], Section 3. Note that "Cache-Control: no-cache" allows a response to be stored, just not reused by a cache; it does not prevent caching (despite its name).

For example, this response cannot be stored or reused by a cache:

HTTP/1.1 200 OK
Content-Type: application/example+xml
Cache-Control: no-store

[content]

When an application has a need to express a lifetime that’s separate from the freshness lifetime, this should be expressed separately, either in the response’s body or in a separate header field. When this happens, the relationship between HTTP caching and that lifetime need to be carefully considered, since the response will be used as long as it is considered fresh.

Like other functions, HTTP caching is generic; it does not have knowledge of the application in use. Therefore, caching extensions need to be backwards-compatible, as per [I-D.ietf-httpbis-cache], Section 5.2.3.

4.10. Application State

Applications MAY use stateful cookies [I-D.ietf-httpbis-rfc6265bis] to identify a client and/or store client-specific data to contextualise requests.

When used, it is important to carefully specify the scoping and use of cookies; if the application exposes sensitive data or capabilities (e.g., by acting as an ambient authority), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

Applications MUST NOT make assumptions about the relationship between separate requests on a single transport connection; doing so breaks many of the assumptions of HTTP as a stateless protocol, and will cause problems in interoperability, security, operability and evolution.
4.11. Client Authentication

Applications MAY use HTTP authentication [I-D.ietf-httpbis-semantics] to identify clients. The Basic authentication scheme [RFC7617] MUST NOT be used unless the underlying transport is authenticated, integrity-protected and confidential (e.g., as provided the "HTTPS" URL scheme, or another using TLS). The Digest scheme [RFC7616] MUST NOT be used unless the underlying transport is similarly secure, or the chosen hash algorithm is not "MD5".

With HTTPS, clients might also be authenticated using certificates [RFC5246].

When used, it is important to carefully specify the scoping and use of authentication; if the application exposes sensitive data or capabilities (e.g., by acting as an ambient authority), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

4.12. Co-Existing with Web Browsing

Even if there is not an intent for an application to be used with a Web browser, its resources will remain available to browsers and other HTTP clients.

This means that all such applications that use HTTP need to consider how browsers will interact with them, particularly regarding security.

For example, if an application’s state can be changed using a POST request, a Web browser can easily be coaxed into cross-site request forgery (CSRF) from arbitrary Web sites.

Or, if content returned from the application’s resources is under control of an attacker (for example, part of the request is reflected in the response, or the response contains external information that might be under the control of the attacker), a cross-site scripting (XSS) attack is possible, whereby an attacker can inject code into the browser and access data and capabilities on that origin.

This is only a small sample of the kinds of issues that applications using HTTP must consider. Generally, the best approach is to consider the application actually as a Web application, and to follow best practices for their secure development.

A complete enumeration of such practices is out of scope for this document, but some considerations include:
Using an application-specific media type in the Content-Type header, and requiring clients to fail if it is not used

Using X-Content-Type-Options: nosniff [FETCH] to assure that content under attacker control can’t be coaxed into a form that is interpreted as active content by a Web browser

Using Content-Security-Policy [CSP] to constrain the capabilities of active content (such as HTML [HTML5]), thereby mitigating Cross-Site Scripting attacks

Using Referrer-Policy [REFERRER-POLICY] to prevent sensitive data in URLs from being leaked in the Referer request header

Using the ‘HttpOnly’ flag on Cookies to assure that cookies are not exposed to browser scripting languages [I-D.ietf-httpbis-rfc6265bis]

Avoiding use of compression on any sensitive information (e.g., authentication tokens, passwords), as the scripting environment offered by Web browsers allows an attacker to repeatedly probe the compression space; if the attacker has access to the path of the communication, they can use this capability to recover that information

Depending on how they are intended to be deployed, specifications for applications using HTTP might require the use of these mechanisms in specific ways, or might merely point them out in Security Considerations.

An example of a HTTP response from an application that does not intend for its content to be treated as active by browsers might look like this:

HTTP/1.1 200 OK
Content-Type: application/example+json
X-Content-Type-Options: nosniff
Content-Security-Policy: default-src ‘none’
Cache-Control: max-age=3600
Referrer-Policy: no-referrer
[content]

If an application has browser compatibility as a goal, client interaction ought to be defined in terms of [FETCH], since that is the abstraction that browsers use for HTTP; it enforces many of these best practices.
4.13. Application Boundaries

Because the origin [RFC6454] is how many HTTP capabilities are scoped, applications also need to consider how deployments might interact with other applications (including Web browsing) on the same origin.

For example, if Cookies [I-D.ietf-httpbis-rfc6265bis] are used to carry application state, they will be sent with all requests to the origin by default, unless scoped by path, and the application might receive cookies from other applications on the origin. This can lead to security issues, as well as collision in cookie names.

One solution to these issues is to require a dedicated hostname for the application, so that it has a unique origin. However, it is often desirable to allow multiple applications to be deployed on a single hostname; doing so provides the most deployment flexibility and enables them to be "mixed" together (See [RFC7320] for details). Therefore, applications using HTTP should strive to allow multiple applications on an origin.

To enable this, when specifying the use of Cookies, HTTP authentication realms [I-D.ietf-httpbis-semantics], or other origin-wide HTTP mechanisms, applications using HTTP SHOULD NOT mandate the use of a particular name, but instead let deployments configure them. Consideration SHOULD be given to scoping them to part of the origin, using their specified mechanisms for doing so.

Modern Web browsers constrain the ability of content from one origin to access resources from another, to avoid leaking private information. As a result, applications that wish to expose cross-origin data to browsers will need to implement the CORS protocol; see [FETCH].


HTTP/2 adds the ability for servers to "push" request/response pairs to clients in [RFC7540], Section 8.2. While server push seems like a natural fit for many common application semantics (e.g., "fanout" and publish/subscribe), a few caveats should be noted:

- Server push is hop-by-hop; that is, it is not automatically forwarded by intermediaries. As a result, it might not work easily (or at all) with proxies, reverse proxies, and Content Delivery Networks.
Server push can have negative performance impact on HTTP when used incorrectly; in particular, if there is contention with resources that have actually been requested by the client.

- Server push is implemented differently in different clients, especially regarding interaction with HTTP caching, and capabilities might vary.

- APIs for server push are currently unavailable in some implementations, and vary widely in others. In particular, there is no current browser API for it.

- Server push is not supported in HTTP/1.1 or HTTP/1.0.

- Server push does not form part of the "core" semantics of HTTP, and therefore might not be supported by future versions of the protocol.

Applications wishing to optimise cases where the client can perform work related to requests before the full response is available (e.g., fetching links for things likely to be contained within) might benefit from using the 103 (Early Hints) status code; see [RFC8297].

Applications using server push directly need to enforce the requirements regarding authority in [RFC7540], Section 8.2, to avoid cross-origin push attacks.

### 4.15. Versioning and Evolution

It’s often necessary to introduce new features into application protocols, and change existing ones.

In HTTP, backwards-incompatible changes are possible using a number of mechanisms:

- Using a distinct link relation type [RFC8288] to identify a URL for a resource that implements the new functionality

- Using a distinct media type [RFC6838] to identify formats that enable the new functionality

- Using a distinct HTTP header field to implement new functionality outside the message body
5. IANA Considerations

This document has no requirements for IANA.

6. Security Considerations

Section 4.10 discusses the impact of using stateful mechanisms in the protocol as ambient authority, and suggests a mitigation.

Section 4.4.2 requires support for ‘https’ URLs, and discourages the use of ‘http’ URLs, to provide authentication, integrity and confidentiality, as well as mitigate pervasive monitoring attacks.

Section 4.12 highlights the implications of Web browsers’ capabilities on applications that use HTTP.

Section 4.13 discusses the issues that arise when applications are deployed on the same origin as Web sites (and other applications).

Section 4.14 highlights risks of using HTTP/2 server push in a manner other than specified.

Applications that use HTTP in a manner that involves modification of implementations - for example, requiring support for a new URL scheme, or a non-standard method - risk having those implementations "fork" from their parent HTTP implementations, with the possible result that they do not benefit from patches and other security improvements incorporated upstream.

6.1. Privacy Considerations

HTTP clients can expose a variety of information to servers. Besides information that’s explicitly sent as part of an application’s operation (for example, names and other user-entered data), and "on the wire" (which is one of the reasons https is recommended in Section 4.4.2), other information can be gathered through less obvious means - often by connecting activities of a user over time.

This includes session information, tracking the client through fingerprinting, and mobile code.

Session information includes things like the IP address of the client, TLS session tickets, Cookies, ETags stored in the client’s cache, and other stateful mechanisms. Applications are advised to avoid using session mechanisms unless they are unavoidable or necessary for operation, in which case these risks needs to be documented. When they are used, implementations should be encouraged to allow clearing such state.
Fingerprinting uses unique aspects of a client’s messages and behaviours to connect disparate requests and connections. For example, the User-Agent request header conveys specific information about the implementation; the Accept-Language request header conveys the users’ preferred language. In combination, a number of these markers can be used to uniquely identify a client, impacting its control over its data. As a result, applications are advised to specify that clients should only emit the information they need to function in requests.

Finally, if an application exposes the ability to run mobile code, great care needs to be taken, since any ability to observe its environment can be used as an opportunity to both fingerprint the client and to obtain and manipulate private data (including session information). For example, access to high-resolution timers (even indirectly) can be used to profile the underlying hardware, creating a unique identifier for the system. Applications are advised avoid allowing the use of mobile code where possible; when it cannot be avoided, the resulting system’s security properties need be carefully scrutinised.

7. References

7.1. Normative References

[I-D.ietf-httpbis-cache]

[I-D.ietf-httpbis-messaging]

[I-D.ietf-httpbis-semantics]


7.2. Informative References


Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", draft-nottingham-rfc5785bis-08 (work in progress), October 2018.


7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

Appendix A. Changes from RFC 3205

[RFC3205] captured the Best Current Practice in the early 2000’s, based on the concerns facing protocol designers at the time. Use of HTTP has changed considerably since then, and as a result this document is substantially different. As a result, the changes are too numerous to list individually.

Author’s Address

Mark Nottingham
Email: mnot@mnot.net
URI: https://www.mnot.net/
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), 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.
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...
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:

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

- "P", an integer smaller than 256, that indicates the probability of a false positive that is acceptable, expressed as "1/2**P".
- "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"*"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:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
- "maxcount" - max number of cuckoo hops
- "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:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
- "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 \(40 + h \times f \times b\).
   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:

- "key", an array of characters
- "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:

- "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:

- "key", an array of characters.
- "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:

- "hash1", an integer indicating the previous hash.
- "fingerprint", an integer indicating the fingerprint value.
- "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;

- 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.

- 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:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234].

- "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 + "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", 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]:

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

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

Internet-Draft    Cache Digests for HTTP/2    July 2018

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]:

- Header field name: Cache-Digest
- Applicable protocol: http
- Status: experimental
- Author/Change controller: IESG
- 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

7.2. Informative References


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 respresents 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

- Remove ETag from the digest key calculations.
- Add SETTINGS_ prefix to parameter names.

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

- Yoav becomes an author; Mark steps down.

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

- Switch to Cuckoo Filter.

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

- Added definition of the Cache-Digest header.
- Introduce ACCEPT_CACHE_DIGEST SETTINGS parameter.
B.5. Since draft-ietf-httpbis-cache-digest-00

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

Appendix C. Acknowledgements

Thanks to Stefan Eissing for his suggestions.

Authors’ Addresses

Kazuho Oku
Fastly
Email: kazuhooku@gmail.com

Yoav Weiss
Akamai
Email: yoav@yoav.ws
URI: https://blog.yoav.ws/
Abstract

This document defines the CDN-Loop request header field for HTTP. CDN-Loop addresses an operational need that occurs when an HTTP request is intentionally forwarded between Content Delivery Networks (CDNs), but is then accidentally or maliciously re-routed back into the original CDN causing a non-terminating loop. The new header field can be used to identify the error and terminate the loop.

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 August 8, 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 (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
1. Introduction

In modern deployments of HTTP servers, it is common to interpose Content Delivery Networks (CDNs) in front of origin servers to improve end-user perceived latency, reduce operational costs, and improve scalability and reliability of services.

Often, more than one CDN is in use by a given origin. This happens for a variety of reasons, such as cost savings, arranging for failover should one CDN have issues, or to directly compare their services.

As a result, it is not unknown for forwarding CDNs to be configured in a "loop" accidentally; because routing is achieved through a combination of DNS and forwarding rules, and site configurations are sometimes complex and managed by several parties.

When this happens, it is difficult to debug. Additionally, it sometimes isn’t accidental; loops between multiple CDNs can be used as an attack vector (e.g., see [loop-attack]), especially if one CDN unintentionally strips the loop detection headers of another.

This specification defines the CDN-Loop HTTP request header field to help detect such attacks and accidents among implementing forwarding CDNs, by disallowing its modification by their customers.
1.1. Relationship to Via

HTTP defines the Via header field in [RFC7230], Section 5.7.1 for "tracking message forwards, avoiding request loops, and identifying the protocol capabilities of senders along the request/response chain."

In theory, Via could be used to identify these loops. However, in practice it is not used in this fashion, because some HTTP servers use Via for other purposes - in particular, some implementations disable some HTTP/1.1 features when the Via header is present.

1.2. Conventions and Definitions

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 specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with a list extension, defined in Section 7 of [RFC7230], that allows for compact definition of comma-separated lists using a '#' operator (similar to how the '*' operator indicates repetition). Additionally, it uses the token, OWS, uri-host and port rules from [RFC7230] and the parameter rule from [RFC7231].

2. The CDN-Loop Request Header Field

The CDN-Loop request header field is intended to help a Content Delivery Network identify when an incoming request has already passed through that CDN’s servers, to detect loops.

CDN-Loop = #cdn-info
cdn-info = cdn-id *( OWS ';' OWS parameter )
cdn-id = ( uri-host [ ':' port ] ) / pseudonym
pseudonym = token

The cdn-id identifies the CDN using either a hostname under its control or a pseudonym. Hostnames are preferred, to help avoid accidental collisions. If a pseudonym is used, unintentional collisions are more likely, and therefore values should be carefully chosen to prevent them; for example, using a well-known value (such as the recognized name of the CDN in question), or a generated value with enough entropy to make collisions unlikely (such as a UUID [RFC4122]).
Optionally, cdn-info can have semicolon-separated key/value parameters, to accommodate additional information for the CDN’s use.

Conforming Content Delivery Networks SHOULD add a cdn-info to this header field in all requests they generate or forward (creating the header field if necessary).

As with all HTTP header fields defined using the "#" rule, the CDN-Loop header field can be added to by comma-separating values, or by creating a new header field with the desired value.

For example:

GET /image.jpg HTTP/1.1
Host: cdn-customer.example
User-Agent: ExampleBrowser/5
CDN-Loop: foo123.foocdn.example, barcdn.example; trace="abcdef"
CDN-Loop: AnotherCDN; abc=123; def="456"

Note that the pseudonym syntax does not allow whitespace, DQUOTE or any of the characters "(),;/:<=>?@[{}". See [RFC7230], Section 3.2.6. Likewise, note the rules for when parameter values need to be quoted in [RFC7231], Section 3.1.1.

The effectiveness of this mechanism relies on all intermediaries preserving the header field, since removing (or allowing it to be removed, e.g., by customer configuration) would prevent downstream CDNs from using it to detect looping. In general, unknown header fields are not removed by intermediaries, but there may be need to add CDN-Loop to an implementation’s list of header fields that are not to be removed under any circumstances. The header field SHOULD NOT be used for other purposes.

3. Security Considerations

The threat model that the CDN-Loop header field addresses is a customer who is attacking a service provider by configuring a forwarding loop by accident or malice. For it to function, CDNs cannot allow customers to modify or remove it in their configuration (see Section 2).

Note that a CDN that allows customers to remove or modify the CDN-Loop header field (i.e., they do not implement this specification) remains an attack vector against both implementing and non-implementing CDNs.
A CDN’s use of the CDN-Loop header field might expose its presence. For example, if CDN A is configured to forward its requests to CDN B for a given origin, CDN B’s presence can be revealed if it behaves differently based upon the presence of the CDN-Loop header field.

The CDN-Loop header field can be generated by any client, and therefore its contents cannot be trusted. CDNs who modify their behaviour based upon its contents should assure that this does not become an attack vector (e.g., for Denial-of-Service).

It is possible to sign the contents of the header field (either by putting the signature directly into the field’s content, or using another header field), but such use is not defined (or required) by this specification.

Depending on how it is used, CDN-Loop can expose information about the internal configuration of the CDN; for example, the number of hops inside the CDN, and the hostnames of nodes.

4. IANA Considerations

This document registers the "CDN-Loop" header field in the Permanent Message Header Field Names registry.

- Header Field Name: CDN-Loop
- Protocol: http
- Status: standard
- Reference: (this document)

5. References

5.1. Normative References


5.2. Informative References

[loop-attack]


Authors’ Addresses

Stephen Ludin
Akamai Technologies
Email: sludin@akamai.com

Mark Nottingham
Fastly
Email: mnot@fastly.com

Nick Sullivan
Cloudflare
Email: nick@cloudflare.com

Ludin, et al. Expires August 8, 2019
HTTP Working Group                                           I. Grigorik
Internet-Draft                                                    Google
Intended status: Experimental                             March 11, 1999
Expires: September 12, 2019

HTTP Client Hints
draft-ietf-httpbis-client-hints-07

Abstract

HTTP defines proactive content negotiation to allow servers to select
the appropriate response for a given request, based upon the user
agent’s characteristics, as expressed in request headers. In
practice, clients are often unwilling to send those request headers,
because it is not clear whether they will be used, and sending them
impacts both performance and privacy.

This document defines two response headers, Accept-CH and Accept-CH-
Lifetime, that servers can use to advertise their use of request
headers for proactive content negotiation, along with a set of
guidelines for the creation of such headers, colloquially known as
"Client Hints."

It also defines an initial set of Client Hints.

Note to Readers

Discussion of this draft takes place on the HTTP working group
mailing list (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

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 September 12, 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 (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. Notational Conventions .......................... 4
2. Client Hint Request Header Fields ...................... 4
   2.1. Sending Client Hints ............................ 4
   2.2. Server Processing of Client Hints .............. 5
       2.2.1. Advertising Support via Accept-CH Header Field ... 5
       2.2.2. The Accept-CH-Lifetime Header Field .......... 5
       2.2.3. Interaction with Caches ...................... 6
3. Security Considerations .............................. 7
4. IANA Considerations ................................ 7
   4.1. Accept-CH ....................................... 8
   4.2. Accept-CH-Lifetime .............................. 8
5. References ........................................... 8
   5.1. Normative References ............................ 8
   5.2. Informative References ........................... 9
   5.3. URIs ............................................ 9
appendix A. Interaction with Key Response Header Field .... 9
appendix B. Changes .................................. 10
   B.1. Since -00 ..................................... 10
   B.2. Since -01 ..................................... 10
   B.3. Since -02 ..................................... 10
   B.4. Since -03 ..................................... 10
   B.5. Since -04 ..................................... 10
   B.6. Since -05 ..................................... 10
   B.7. Since -06 ..................................... 10
   B.8. Since -07 ..................................... 11
1. Introduction

There are thousands of different devices accessing the web, each with different device capabilities and preference information. These device capabilities include hardware and software characteristics, as well as dynamic user and client preferences.

One way to infer some of these capabilities is through User-Agent (Section 5.5.3 of [RFC7231]) header field detection against an established database of client signatures. However, this technique requires acquiring such a database, integrating it into the serving path, and keeping it up to date. However, even once this infrastructure is deployed, user agent sniffing has numerous limitations:

- User agent detection cannot reliably identify all static variables
- User agent detection cannot infer any dynamic client preferences
- User agent detection requires an external device database
- User agent detection is not cache friendly

A popular alternative strategy is to use HTTP cookies ([RFC6265]) to communicate some information about the user agent. However, this approach is also not cache friendly, bound by same origin policy, and often imposes additional client-side latency by requiring JavaScript execution to create and manage HTTP cookies.

Proactive content negotiation (Section 3.4.1 of [RFC7231]) offers an alternative approach; user agents use specified, well-defined request headers to advertise their capabilities and characteristics, so that servers can select (or formulate) an appropriate response.

However, proactive content negotiation requires clients to send these request headers prolifically. This causes performance concerns (because it creates "bloat" in requests), as well as privacy issues; passively providing such information allows servers to silently fingerprint the user agent.

This document defines a new response header, Accept-CH, that allows an origin server to explicitly ask that clients send these headers in requests, for a period of time bounded by the Accept-CH-Lifetime response header. It also defines guidelines for content negotiation mechanisms that use it, colloquially referred to as Client Hints.

Client Hints mitigate the performance concerns by assuring that clients will only send the request headers when they’re actually
going to be used, and the privacy concerns of passive fingerprinting by requiring explicit opt-in and disclosure of required headers by the server through the use of the Accept-CH response header.

This document defines the Client Hints infrastructure, a framework that enables servers to opt-in to specific proactive content negotiation features, which will enable them to adapt their content accordingly. However, it does not define any specific features that will use that infrastructure. Those features will be defined in their respective specifications.

This document does not supersede or replace the User-Agent header field. Existing device detection mechanisms can continue to use both mechanisms if necessary. By advertising user agent capabilities within a request header field, Client Hints allow for cache friendly and proactive content negotiation.

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.

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with the list rule extension defined in [RFC7230], Appendix B. It includes by reference the DIGIT rule from [RFC5234] and the OWS and field-name rules from [RFC7230].

2. Client Hint Request Header Fields

A Client Hint request header field is a HTTP header field that is used by HTTP clients to indicate configuration data that can be used by the server to select an appropriate response. Each one conveys client preferences that the server can use to adapt and optimize the response.

2.1. Sending Client Hints

Clients control which Client Hints are sent in requests, based on their default settings, user configuration, and server preferences. The client and server can use an opt-in mechanism outlined below to negotiate which fields should be sent to allow for efficient content adaption, and optionally use additional mechanisms to negotiate delegation policies that control access of third parties to same fields.
Implementers should be aware of the passive fingerprinting implications when implementing support for Client Hints, and follow the considerations outlined in "Security Considerations" section of this document.

2.2. Server Processing of Client Hints

When presented with a request that contains one or more client hint header fields, servers can optimize the response based upon the information in them. When doing so, and if the resource is cacheable, the server MUST also generate a Vary response header field (Section 7.1.4 of [RFC7231]) to indicate which hints can affect the selected response and whether the selected response is appropriate for a later request.

Further, depending on the hint used, the server can generate additional response header fields to convey related values to aid client processing.

2.2.1. Advertising Support via Accept-CH Header Field

Servers can advertise support for Client Hints using the Accept-CH header field or an equivalent HTML meta element with http-equiv attribute ([HTML5]).

Accept-CH = #field-name

For example:

Accept-CH: Sec-CH-Example, Sec-CH-Example-2

When a client receives an HTTP response advertising support for Client Hints, it should process it as origin ([RFC6454]) opt-in to receive Client Hint header fields advertised in the field-value. The opt-in MUST be delivered over a secure transport.

For example, based on Accept-CH example above, a user agent could append the Sec-CH-Example and Sec-CH-Example-2 header fields to all same-origin resource requests initiated by the page constructed from the response.

2.2.2. The Accept-CH-Lifetime Header Field

Servers can ask the client to remember the set of Client Hints that the server supports for a specified period of time, to enable delivery of Client Hints on subsequent requests to the server’s origin ([RFC6454]).
Accept-CH-Lifetime = #delta-seconds

When a client receives an HTTP response that contains Accept-CH-Lifetime header field, the field-value indicates that the Accept-CH preference SHOULD be persisted and bound to the origin, and be considered stale after response’s age ([RFC7234], section 4.2) is greater than the specified number of seconds. The preference MUST be delivered over a secure transport, and MUST NOT be persisted for an origin that isn’t HTTPS.

Accept-CH: Sec-CH-Example, Sec-CH-Example-2
Accept-CH: Sec-CH-Example-3
Accept-CH-Lifetime: 86400

For example, based on the Accept-CH and Accept-CH-Lifetime example above, which is received in response to a user agent navigating to "https://example.com", and delivered over a secure transport: a user agent SHOULD persist an Accept-CH preference bound to "https://example.com" for up to 86400 seconds (1 day), and use it for user agent navigations to "https://example.com" and any same-origin resource requests initiated by the page constructed from the navigation’s response. This preference SHOULD NOT extend to resource requests initiated to "https://example.com" from other origins.

If Accept-CH-Lifetime occurs in a message more than once, the last value overrides all previous occurrences.

2.2.3. Interaction with Caches

When selecting an optimized response based on one or more Client Hints, and if the resource is cacheable, the server needs to generate a Vary response header field ([RFC7234]) to indicate which hints can affect the selected response and whether the selected response is appropriate for a later request.

Vary: Sec-CH-Example

Above example indicates that the cache key needs to include the Sec-CH-Example header field.

Vary: Sec-CH-Example, Sec-CH-Example-2

Above example indicates that the cache key needs to include the Sec-CH-Example and Sec-CH-Example-2 header fields.
3. Security Considerations

The request header fields defined in this document, and those that extend it, expose information about the user’s environment to enable proactive content negotiation. Such information may reveal new information about the user and implementers ought to consider the following considerations, recommendations, and best practices.

Transmitted Client Hints header fields SHOULD NOT provide new information that is otherwise not available to the application via other means, such as using HTML, CSS, or JavaScript. Further, sending highly granular data, such as image and viewport width may help identify users across multiple requests. Reducing the set of field values that can be expressed, or restricting them to an enumerated range where the advertised value is close but is not an exact representation of the current value, can improve privacy and reduce risk of linkability by ensuring that the same value is sent by multiple users. However, such precautions can still be insufficient for some types of data, especially data that can change over time.

Implementers ought to consider both user and server controlled mechanisms and policies to control which Client Hints header fields are advertised:

- Implementers SHOULD restrict delivery of some or all Client Hints header fields to the opt-in origin only, unless the opt-in origin has explicitly delegated permission to another origin to request Client Hints header fields.
- Implementers MAY provide user choice mechanisms so that users may balance privacy concerns with bandwidth limitations. However, implementers should also be aware that explaining the privacy implications of passive fingerprinting to users may be challenging.
- Implementations specific to certain use cases or threat models MAY avoid transmitting some or all of Client Hints header fields. For example, avoid transmission of header fields that can carry higher risks of linkability.

Implementers SHOULD support Client Hints opt-in mechanisms and MUST clear persisted opt-in preferences when any one of site data, browsing history, browsing cache, or similar, are cleared.

4. IANA Considerations

This document defines the "Accept-CH" and "Accept-CH-Lifetime" HTTP response fields, and registers them in the Permanent Message Header Fields registry.
4.1. Accept-CH

- Header field name: Accept-CH
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 2.2.1 of this document
- Related information: for Client Hints

4.2. Accept-CH-Lifetime

- Header field name: Accept-CH-Lifetime
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 2.2.2 of this document
- Related information: for Client Hints

5. References

5.1. Normative References


5.2. Informative References


Appendix A. Interaction with Key Response Header Field

Client Hints may be combined with Key response header field ([KEY]) to enable fine-grained control of the cache key for improved cache efficiency. For example, the server can return the following set of instructions:

Key: Sec-CH-Example;partition=1.5:2.5:4.0

Above example indicates that the cache key needs to include the value of the Sec-CH-Example header field with three segments: less than 1.5, 1.5 to less than 2.5, and 4.0 or greater.

Key: Width;Sec-CH-Example=320
Above example indicates that the cache key needs to include the value of the Sec-CH-Example header field and be partitioned into groups of 320: 0-320, 320-640, and so on.

Appendix B. Changes

B.1. Since -00

- Issue 168 (make Save-Data extensible) updated ABNF.
- Issue 163 (CH review feedback) editorial feedback from httpwg list.
- Issue 153 (NetInfo API citation) added normative reference.

B.2. Since -01

- Issue 200: Moved Key reference to informative.
- Issue 215: Extended passive fingerprinting and mitigation considerations.
- Changed document status to experimental.

B.3. Since -02

- Issue 239: Updated reference to CR-css-values-3
- Issue 240: Updated reference for Network Information API
- Issue 241: Consistency in IANA considerations
- Issue 250: Clarified Accept-CH

B.4. Since -03

- Issue 284: Extended guidance for Accept-CH
- Issue 308: Editorial cleanup
- Issue 306: Define Accept-CH-Lifetime

B.5. Since -04

- Issue 361: Removed Downlink
- Issue 361: Moved Key to appendix, plus other editorial feedback

B.6. Since -05

- Issue 372: Scoped CH opt-in and delivery to secure transports
- Issue 373: Bind CH opt-in to origin

B.7. Since -06

- Issue 524: Save-Data is now defined by NetInfo spec, dropping
B.8. Since -07

  o Removed specific features to be defined in other specifications

Acknowledgements

Thanks to Mark Nottingham, Julian Reschke, Chris Bentzel, Yoav Weiss, Ben Greenstein, Tarun Bansal, Roy Fielding, Vasily Faronov, Ted Hardie, Jonas Sicking, and numerous other members of the IETF HTTP Working Group for invaluable help and feedback.

Author’s Address

Ilya Grigorik
Google

Email: ilya@igvita.com
URI: https://www.igvita.com/
Abstract

This document describes a set of data types and algorithms associated with them that are intended to make it easier and safer to define and handle HTTP header fields. It is intended for use by new specifications of HTTP header fields as well as revisions of existing header field specifications when doing so does not cause interoperability issues.

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), 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-header-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 October 19, 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 (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. Intentionally Strict Processing .......................... 4
   1.2. Notational Conventions .................................. 4
2. Defining New Structured Headers ............................... 5
3. Structured Header Data Types .................................. 7
   3.1. Dictionaries .............................................. 7
   3.2. Lists .................................................... 7
   3.3. Lists of Lists ........................................... 8
   3.4. Parameterised Lists ....................................... 8
   3.5. Items ..................................................... 9
   3.6. Integers ................................................ 9
   3.7. Floats ................................................... 9
   3.8. Strings .................................................. 10
   3.9. Tokens .................................................. 11
   3.10. Byte Sequences .......................................... 11
   3.11. Booleans ............................................... 11
4. Structured Headers in HTTP/1 .................................. 12
   4.1. Serialising Structured Headers into HTTP/1 .............. 12
   4.2. Parsing HTTP/1 Header Fields into Structured Headers ..... 18
5. IANA Considerations ........................................... 27
6. Security Considerations ....................................... 28
7. References .................................................... 28
   7.1. Normative References ..................................... 28
   7.2. Informative References ................................. 29
1. Introduction

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

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

This document introduces a set of common data structures for use in HTTP header field values to address these problems. In particular, it defines a generic, abstract model for header field values, along with a concrete serialisation for expressing that model in HTTP/1 [RFC7230] header fields.

HTTP headers that are defined as "Structured Headers" use the types defined in this specification to define their syntax and basic handling rules, thereby simplifying both their definition by specification writers and handling by implementations.

Additionally, future versions of HTTP can define alternative serialisations of the abstract model of these structures, allowing headers that use it 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 headers; the mechanisms described herein are only intended to be used with headers that explicitly opt into them.

To specify a header field that is a Structured Header, see Section 2.

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

Those abstract types can be serialised into and parsed from textual headers - such as those used in HTTP/1 - using the algorithms described in Section 4.

1.1. Intentionally Strict Processing

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

This is designed to encourage faithful implementation and therefore good interoperability. Therefore, implementations that try to be "helpful" by being more tolerant of input are doing a disservice to the overall community, since it will encourage 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 early, and avoids both interoperability and security issues that might otherwise result.

Note that as a result of this strictness, if a header field is appended to by multiple parties (e.g., intermediaries, or different components in the sender), it could be that an error in one party’s value causes the entire header field 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 the Augmented Backus-Naur Form (ABNF) notation of [RFC5234], including the VCHAR, SP, DIGIT, ALPHA and DQUOTE rules from that document. It also includes the OWS rule from [RFC7230].
This document uses algorithms to specify parsing and serialisation behaviours, and ABNF to illustrate expected syntax in HTTP/1-style header fields.

For parsing from HTTP/1 header fields, implementations MUST follow the algorithms, but MAY vary in implementation so as the behaviours are indistinguishable from specified behaviour. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence. In some places, the algorithms are "greedy" with whitespace, but this should not affect conformance.

For serialisation to HTTP/1 header fields, the ABNF illustrates the range of acceptable wire representations with as much fidelity as possible, and the algorithms define the recommended way to produce them. Implementations MAY vary from the specified behaviour so long as the output still matches the ABNF.

2. Defining New Structured Headers

To define a HTTP header as a structured header, its specification needs to:

- Reference this specification. Recipients and generators of the header need to know that the requirements of this document are in effect.

- Specify the header field’s allowed syntax for values, in terms of the types described in Section 3, along with their associated semantics. Syntax definitions are encouraged to use the ABNF rules beginning with "sh-" defined in this specification.

- Specify any additional constraints upon the syntax of the structured used, as well as the consequences when those constraints are violated. When Structured Headers parsing fails, the header is discarded (see Section 4.2); in most situations, header-specific constraints should do likewise.

Note that a header field definition cannot relax the requirements of a structure or its processing because doing so would preclude handling by generic software; they can only add additional constraints. Likewise, header field definitions should use Structured Headers for the entire header field value, not a portion thereof.

For example:
# Foo-Example Header

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

Foo-Example is a Structured Header [RFCxxxx]. Its value MUST be a dictionary ([RFCxxxx], Section Y.Y). Its ABNF is:

```
Foo-Example = sh-dictionary
```

The dictionary MUST contain:

* Exactly one member whose key is "foo", and whose value is an integer ([RFCxxxx], Section Y.Y), indicating the number of foos in the message.
* Exactly one member whose key is "barUrls", and whose value is a string ([RFCxxxx], Section Y.Y), conveying the Bar URLs for the message. See below for processing requirements.

If the parsed header field does not contain both, it MUST be ignored.

"foo" MUST be between 0 and 10, inclusive; other values MUST cause the header to be ignored.

"barUrls" contains a space-separated list of URI-references ([RFC3986], Section 4.1):

```
barURLs = URI-reference *( 1*SP URI-reference )
```

If a member of barURLs is not a valid URI-reference, it MUST cause that value to be ignored.

If a member of barURLs is a relative reference ([RFC3986], Section 4.2), it MUST be resolved ([RFC3986], Section 5) before being used.

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

This section defines the abstract value types that can be composed into Structured Headers. The ABNF provided represents the on-wire format in HTTP/1.

3.1. Dictionaries

Dictionaries are ordered maps of key-value pairs, where the keys are short, textual strings and the values are items (Section 3.5). There can be one or more members, and keys are required to be unique.

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

The ABNF for dictionaries in HTTP/1 headers is:

```
sh-dictionary  = dict-member *( OWS "," OWS dict-member )
dict-member    = member-name ";" member-value
member-name    = key
member-value   = sh-item
key             = lcalpha *( lcalpha / DIGIT / "_" / "-" )
lcalpha        = %x61-7A ; a-z
```

In HTTP/1, keys and values are separated by ";" (without whitespace), and key/value pairs are separated by a comma with optional whitespace. For example:

```
Example-DictHeader: en="Applepie", da=*w4ZibGV0w6ZydGU=* 
```

Typically, a header field specification will define the semantics of individual keys, as well as whether their presence is required or optional. Recipients MUST ignore keys that are undefined or unknown, unless the header field’s specification specifically disallows them.

Parsers MUST support dictionaries containing at least 1024 key/value pairs, and dictionary keys with at least 64 characters.

3.2. Lists

Lists are arrays of items (Section 3.5) with one or more members.

The ABNF for lists in HTTP/1 headers is:

```
sh-list     = list-member *( OWS "," OWS list-member )
list-member = sh-item
```
Internet-Draft

Structured Headers for HTTP

April 2019

In HTTP/1, each member is separated by a comma and optional
whitespace. For example, a header field whose value is defined as a
list of strings could look like:
Example-StrListHeader: "foo", "bar", "It was the best of times."
Header specifications can constrain the types of individual values if
necessary.
Parsers MUST support lists containing at least 1024 members.
3.3.

Lists of Lists

Lists of Lists are arrays of arrays containing items (Section 3.5).
The ABNF for lists of lists in HTTP/1 headers is:
sh-listlist = inner-list *( OWS "," OWS inner-list )
inner-list = list-member *( OWS ";" OWS list-member )
In HTTP/1, each inner-list is separated by a comma and optional
whitespace, and members of the inner-list are separated by semicolons
and optional whitespace. For example, a header field whose value is
defined as a list of lists of strings could look like:
Example-StrListListHeader: "foo";"bar", "baz", "bat"; "one"
Header specifications can constrain the types of individual innerlist values if necessary.
Parsers MUST support lists of lists containing at least 1024 members,
and inner-lists containing at least 256 members.
3.4.

Parameterised Lists

Parameterised Lists are arrays of parameterised identifiers, with one
or more members.
A parameterised identifier is a primary identifier (a Section 3.9})
with associated parameters, an ordered map of key-value pairs where
the keys are short, textual strings and the values are items
(Section 3.5). There can be zero or more parameters, and keys are
required to be unique.
The ABNF for parameterised lists in HTTP/1 headers is:

Nottingham & Kamp

Expires October 19, 2019

[Page 8]


In HTTP/1, each param-id is separated by a comma and optional whitespace (as in Lists), and the parameters are separated by semicolons. For example:

```
Example-ParamListHeader: abc_123;a=1;b=2; cdef_456, ghi;q="9";r="w"
```

Parsers MUST support parameterised lists containing at least 1024 members, support members with at least 256 parameters, and support parameter keys with at least 64 characters.

### 3.5. Items

An item is can be an integer (Section 3.6), float (Section 3.7), string (Section 3.8), token (Section 3.9), byte sequence (Section 3.10), or Boolean (Section 3.11).

The ABNF for items in HTTP/1 headers is:

```plaintext
sh-item = sh-integer / sh-float / sh-string / sh-token / sh-binary / sh-boolean
```

### 3.6. 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).

The ABNF for integers in HTTP/1 headers is:

```plaintext
sh-integer = ["-"] 1*15DIGIT
```

For example:

```
Example-IntegerHeader: 42
```

### 3.7. Floats

Floats are integers with a fractional part, that can be stored as IEEE 754 double precision numbers (binary64) ([IEEE754]).

The ABNF for floats in HTTP/1 headers is:
sh-float = ["-" ] ( 
  DIGIT \.* 1*14DIGIT / 
  2DIGIT \.* 1*13DIGIT / 
  3DIGIT \.* 1*12DIGIT / 
  4DIGIT \.* 1*11DIGIT / 
  5DIGIT \.* 1*10DIGIT / 
  6DIGIT \.* 1*9DIGIT / 
  7DIGIT \.* 1*8DIGIT / 
  8DIGIT \.* 1*7DIGIT / 
  9DIGIT \.* 1*6DIGIT / 
  10DIGIT \.* 1*5DIGIT / 
  11DIGIT \.* 1*4DIGIT / 
  12DIGIT \.* 1*3DIGIT / 
  13DIGIT \.* 1*2DIGIT / 
  14DIGIT \.* 1DIGIT )

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

Example-FloatHeader: 4.5

3.8. Strings

Strings are zero or more printable ASCII [RFC0020] characters (i.e., the range 0x20 to 0x7E). Note that this excludes tabs, newlines, carriage returns, etc.

The ABNF for strings in HTTP/1 headers is:

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

In HTTP/1 headers, strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

Example-StringHeader: "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 sequences MUST cause parsing to fail.

Unicode is not directly supported in this document, because it causes a number of interoperability issues, and - with few exceptions - header values do not require it.
When it is necessary for a field value to convey non-ASCII string content, a byte sequence (Section 3.10) SHOULD be specified, along with a character encoding (preferably UTF-8).

Parsers MUST support strings with at least 1024 characters.

### 3.9. Tokens

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

The ABNF for tokens in HTTP/1 headers is:

```
sh-token = ALPHA *( ALPHA / DIGIT / "_" / "-" / "." / ":" / "+" / "+" / "/" )
```

Parsers MUST support tokens with at least 512 characters.

Note that a Structured Header token is not the same as the "token" ABNF rule defined in [RFC7230].

### 3.10. Byte Sequences

Byte sequences can be conveyed in Structured Headers.

The ABNF for a byte sequence in HTTP/1 headers is:

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

In HTTP/1 headers, a byte sequence is delimited with asterisks and encoded using base64 ([RFC4648], Section 4). For example:

Example-BinaryHdr: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==*

Parsers MUST support byte sequences with at least 16384 octets after decoding.

### 3.11. Booleans

Boolean values can be conveyed in Structured Headers.

The ABNF for a Boolean in HTTP/1 headers is:

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

In HTTP/1 headers, a boolean is indicated with a leading "?" character. For example:
4. Structured Headers in HTTP/1

This section defines how to serialise and parse Structured Headers in HTTP/1 textual header fields, and protocols compatible with them (e.g., in HTTP/2 [RFC7540] before HPACK [RFC7541] is applied).

4.1. Serialising Structured Headers into HTTP/1

Given a structured defined in this specification:

1. If the structure is a dictionary, return the result of Serialising a Dictionary (Section 4.1.1).
2. If the structure is a parameterised list, return the result of Serialising a Parameterised List (Section 4.1.4).
3. If the structure is a list of lists, return the result of Serialising a List of Lists (\(\text{ser-listlist}\)).
4. If the structure is a list, return the result of Serialising a List (Section 4.1.2).
5. If the structure is an item, return the result of Serialising an Item (Section 4.1.5).
6. Otherwise, fail serialisation.

4.1.1. Serialising a Dictionary

Given a dictionary as input_dictionary:

1. Let output be an empty string.
2. For each member mem of input_dictionary:
   1. Let name be the result of applying Serialising an Key (Section 4.1.1.1) to mem’s member-name.
   2. Append name to output.
   3. Append "=" to output.
   4. Let value be the result of applying Serialising an Item (Section 4.1.5) to mem’s member-value.
   5. Append value to output.
6. If more members remain in input_dictionary:
   1. Append a COMMA to output.
   2. Append a single WS to output.
   3. Return output.

4.1.1.1. Serialising a Key

   Given a key as input_key:
   1. If input_key is not a sequence of characters, or contains characters not allowed in the ABNF for key, fail serialisation.
   2. Let output be an empty string.
   3. Append input_key to output, using ASCII encoding [RFC0020].
   4. Return output.

4.1.2. Serialising a List

   Given a list as input_list:
   1. Let output be an empty string.
   2. For each member mem of input_list:
      1. Let value be the result of applying Serialising an Item (Section 4.1.5) to mem.
      2. Append value to output.
      3. If more members remain in input_list:
         1. Append a COMMA to output.
         2. Append a single WS to output.
   3. Return output.

4.1.3. Serialising a List of Lists

   Given a list of lists of items as input_list:
   1. Let output be an empty string.
2. For each member inner_list of input_list:
   1. If inner_list is not a list, fail serialisation.
   2. If inner_list is empty, fail serialisation.
   3. For each inner_mem of inner_list:
      1. Let value be the result of applying Serialising an Item
         (Section 4.1.5) to inner_mem.
      2. Append value to output.
      3. If more members remain in inner_list:
         1. Append a ";" to output.
         2. Append a single WS to output.
      4. If more members remain in input_list:
         1. Append a COMMA to output.
         2. Append a single WS to output.
   3. Return output.

4.1.4. Serialising a Parameterised List

Given a parameterised list as input_plist:

1. Let output be an empty string.
2. For each member mem of input_plist:
   1. Let id be the result of applying Serialising a Token
      (Section 4.1.9) to mem’s token.
   2. Append id to output.
   3. For each parameter in mem’s parameters:
      1. Append ";" to output.
      2. Let name be the result of applying Serialising a Key
         (Section 4.1.1.1) to parameter’s param-name.
      3. Append name to output.
4. If parameter has a param-value:
   1. Let value be the result of applying Serialising an Item (Section 4.1.5) to parameter’s param-value.
   2. Append "=" to output.
   3. Append value to output.

4. If more members remain in input_plist:
   1. Append a COMMA to output.
   2. Append a single WS to output.

3. Return output.

4.1.5. Serialising an Item

Given an item as input_item:

1. If input_item is an integer, return the result of applying Serialising an Integer (Section 4.1.6) to input_item.

2. If input_item is a float, return the result of applying Serialising a Float (Section 4.1.7) to input_item.

3. If input_item is a string, return the result of applying Serialising a String (Section 4.1.8) to input_item.

4. If input_item is a token, return the result of Serialising a Token (Section 4.1.9) to input_item.

5. If input_item is a Boolean, return the result of applying Serialising a Boolean (Section 4.1.11) to input_item.

6. If input_item is a byte sequence, return the result of applying Serialising a Byte Sequence (Section 4.1.10) to input_item.

7. Otherwise, fail serialisation.

4.1.6. Serialising an Integer

Given an integer as input_integer:

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 serialisation.
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.

4.1.7. Serialising a Float

Given a float as input_float:

1. If input_float is not a IEEE 754 double precision number, fail serialisation.

2. Let output be an empty string.

3. If input_float is less than (but not equal to) 0, append "-" to output.

4. Append input_float’s integer component represented in base 10 using only decimal digits to output; if it is zero, append "0".

5. Append "." to output.

6. Append input_float’s decimal component represented in base 10 using only decimal digits to output; if it is zero, append "0".

7. Return output.

4.1.8. Serialising a String

Given a string as input_string:

1. If input_string is not a sequence of characters, or contains characters outside the range allowed by VCHAR or SP, fail serialisation.

2. Let output be an empty string.

3. Append DQUOTE to output.

4. For each character char in input_string:

   1. If char is "\" or DQUOTE:
1. Append "\" to output.
2. Append char to output, using ASCII encoding [RFC0020].
3. Append DQUOTE to output.
4. Return output.

4.1.9. Serialising a Token

Given a token as input_token:
1. If input_token is not a sequence of characters, or contains characters not allowed in Section 3.9), fail serialisation.
2. Let output be an empty string.
3. Append input_token to output, using ASCII encoding [RFC0020].
4. Return output.

4.1.10. Serialising a Byte Sequence

Given a byte sequence as input_bytes:
1. If input_bytes is not a sequence of bytes, fail serialisation.
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.
4.1.11. Serialising a Boolean

Given a Boolean as input_boolean:

1. If input_boolean is not a boolean, fail serialisation.
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 HTTP/1 Header Fields into Structured Headers

When a receiving implementation parses textual HTTP header fields (e.g., in HTTP/1 or HTTP/2) that are known to be Structured Headers, 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 ASCII string input_string that represents the chosen header's field-value, and header_type, one of "dictionary", "list", "list-list", "param-list", or "item", return the parsed header value.

1. Discard any leading OWS from input_string.
2. If header_type is "dictionary", let output be the result of Parsing a Dictionary from Text (Section 4.2.1).
3. If header_type is "list", let output be the result of Parsing a List from Text (Section 4.2.3).
4. If header_type is "list-list", let output be the result of Parsing a List of Lists from Text (Section 4.2.4).
5. If header_type is "param-list", let output be the result of Parsing a Parameterised List from Text (Section 4.2.5).
6. If header_type is "item", let output be the result of Parsing an Item from Text (Section 4.2.7).
7. Discard any leading OWS from input_string.
8. If input_string is not empty, fail parsing.
9. Otherwise, return output.

When generating input_string, parsers MUST combine all instances of the target header field into one comma-separated field-value, as per [RFC7230], Section 3.2.2; this assures that the header is processed correctly.

For Lists, Lists of Lists, Parameterised Lists and Dictionaries, this has the effect of correctly concatenating all instances of the header field, as long as individual members of the top-level data structure are not split across multiple header instances.

Strings split across multiple header instances 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 serialiser or the parser.

Integers, Floats and Byte Sequences cannot be split across multiple headers because the inserted commas will cause parsing to fail.

If parsing fails - including when calling another algorithm - the entire header field’s value MUST be discarded. This is intentionally strict, to improve interoperability and safety, and specifications referencing this document cannot loosen this requirement.

Note that this has the effect of discarding any header field with non-ASCII characters in input_string.

4.2.1. Parsing a Dictionary from Text

Given an ASCII string input_string, return an ordered map of (key, item). 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 Parse a Key from Text (Section 4.2.2) with input_string.
   2. If dictionary already contains this_key, fail parsing.
   3. Consume the first character of input_string; if it is not "=" fail parsing.
   4. Let this_value be the result of running Parse Item from Text (Section 4.2.7) with input_string.
5. Add key this_key with value this_value to dictionary.
6. Discard any leading OWS from input_string.
7. If input_string is empty, return dictionary.
8. Consume the first character of input_string; if it is not COMMA, fail parsing.
9. Discard any leading OWS from input_string.
10. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.2. Parsing a Key from Text

Given an ASCII string 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, fail parsing.
2. Let output_string be an empty string.
3. While input_string be an empty string.
4. While input_string is not empty:
   1. Let char be the result of removing the first character of input_string.
   2. If char is not one of lcalpha, DIGIT, ",", or "-":
      1. Prepend char to input_string.
      2. Return output_string.
   3. Append char to output_string.
4. Return output_string.

4.2.3. Parsing a List from Text

Given an ASCII string input_string, return a list of items. input_string is modified to remove the parsed value.

1. Let items be an empty array.
2. While input_string is not empty:
1. Let item be the result of running Parse Item from Text (Section 4.2.7) with input_string.

2. Append item to items.

3. Discard any leading OWS from input_string.

4. If input_string is empty, return items.

5. Consume the first character of input_string; if it is not COMMA, fail parsing.

6. Discard any leading OWS from input_string.

7. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.4. Parsing a List of Lists from Text

Given an ASCII string input_string, return a list of lists of items. input_string is modified to remove the parsed value.

1. let top_list be an empty array.

2. Let inner_list be an empty array.

3. While input_string is not empty:

   1. Let item be the result of running Parse Item from Text (Section 4.2.7) with input_string.

   2. Append item to inner_list.

   3. Discard any leading OWS from input_string.

   4. If input_string is empty, append inner_list to top_list and return top_list.

   5. Let char be the result of consuming the first character of input_string.

   6. If char is COMMA:

      1. Append inner_list to top_list.

      2. Let inner_list be an empty array.
7. Else if char is not ";", fail parsing.
8. Discard any leading OWS from input_string.
9. If input_string is empty, fail parsing.

4. No structured data has been found; fail parsing.

4.2.5. Parsing a Parameterised List from Text

Given an ASCII string input_string, return a list of parameterised identifiers. input_string is modified to remove the parsed value.

1. Let items be an empty array.
2. While input_string is not empty:
   1. Let item be the result of running Parse Parameterised Identifier from Text (Section 4.2.6) with input_string.
   2. Append item to items.
   3. Discard any leading OWS from input_string.
   4. If input_string is empty, return items.
   5. Consume the first character of input_string; if it is not COMMA, fail parsing.
   6. Discard any leading OWS from input_string.
   7. If input_string is empty, fail parsing.
3. No structured data has been found; fail parsing.

4.2.6. Parsing a Parameterised Identifier from Text

Given an ASCII string input_string, return a token with an unordered map of parameters. input_string is modified to remove the parsed value.

1. Let primary_identifier be the result of Parsing a Token from Text (Section 4.2.10) from input_string.
2. Let parameters be an empty, ordered map.
3. In a loop:
1. Discard any leading OWS from input_string.
2. If the first character of input_string is not ";", exit the loop.
3. Consume a ";" character from the beginning of input_string.
4. Discard any leading OWS from input_string.
5. Let param_name be the result of Parsing a key from Text (Section 4.2.2) from input_string.
6. If param_name is already present in parameters, fail parsing.
7. Let param_value be a null value.
8. 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 Parsing an Item from Text (Section 4.2.7) from input_string.
9. Add key param_name with value param_value to parameters.
4. Return the tuple (primary_identifier, parameters).

4.2.7. Parsing an Item from Text

Given an ASCII string input_string, return an item. input_string is modified to remove the parsed value.

1. If the first character of input_string is a "-" or a DIGIT, process input_string as a number (Section 4.2.8) and return the result.
2. If the first character of input_string is a DQUOTE, process input_string as a string (Section 4.2.9) and return the result.
3. If the first character of input_string is "*", process input_string as a byte sequence (Section 4.2.11) and return the result.
4. If the first character of input_string is "?", process input_string as a Boolean (Section 4.2.12) and return the result.
5. If the first character of input_string is an ALPHA, process
   input_string as a token (Section 4.2.10) and return the result.

6. Otherwise, fail parsing.

4.2.8. Parsing a Number from Text

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

NOTE: This algorithm parses both Integers Section 3.6 and Floats
Section 3.7, 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 "-", remove it from
   input_string and set sign to -1.
5. If input_string is empty, 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 removing 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 ".", append char to
      input_number and set type to "float".
   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 "float" and input_number contains more than 16
      characters, fail parsing.
8. If type is "integer":

Nottingham & Kamp       Expires October 19, 2019               [Page 24]
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 defined in Section 3.6, fail parsing.

9. Otherwise:
   1. If the final character of input_number is ".", fail parsing.
   2. Parse input_number as a float and let output_number be the product of the result and sign.

10. Return output_number.

4.2.9. Parsing a String from Text

Given an ASCII string 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 removing the first character of input_string.
   2. If char is a backslash ("\"):
      1. If input_string is now empty, fail parsing.
      2. Else:
         1. Let next_char be the result of removing the first character of input_string.
         2. If next_char is not DQUOTE or "\", fail parsing.
         3. 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.

4.2.10. Parsing a Token from Text

Given an ASCII string 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, fail parsing.
2. Let output_string be an empty string.
3. While input_string is not empty:
   1. Let char be the result of removing the first character of input_string.
   2. If char is not one of ALPHA, DIGIT, ":", ":", ":", ":", ":", ":", ":", ":", ":", ":", ":", ":", ":":
      1. Prepend char to input_string.
      2. Return output_string.
   3. Append char to output_string.
4. Return output_string.

4.2.11. Parsing a Byte Sequence from Text

Given an ASCII string 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 removing 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, synthesising padding if necessary (note the 
requirements about recipient behaviour below).

8. Return binary_content.

Because some implementations of base64 do not allow reject of encoded 
data that is not properly ":" padded (see [RFC4648], Section 3.2), 
parsers SHOULD NOT fail when it 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 it is 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.

4.2.12. Parsing a Boolean from Text

Given an ASCII string 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 draft has no actions for IANA.
6. Security Considerations

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

It is possible for parties with the ability to inject new HTTP header fields to change the meaning of a Structured Header. 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


7.2. Informative References

See also http://grouper.ieee.org/groups/754/ [6].


7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/
Appendix A. Acknowledgements

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

Appendix B. Frequently Asked Questions

B.1. Why not JSON?

Earlier proposals for structured headers 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 header 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.

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

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

B.2. Structured Headers don’t "fit" my data.

Structured headers intentionally limits the complexity of data structures, to assure that it can be processed in a performant manner.
with little overhead. This means that work is necessary to fit some
data types into them.

Sometimes, this can be achieved by creating limited substructures in
to values, and/or using more than one header. For example, consider:

Example-Thing: name="Widget", cost=89.2, descriptions="foo bar"
Example-Description: foo; url="https://example.net"; context=123,
bar; url="https://example.org"; context=456

Since the description contains a list of key/value pairs, we use a
Parameterised List to represent them, with the token for each item in
the list used to identify it in the "descriptions" member of the
Example-Thing header.

When specifying more than one header, it’s important to remember to
describe what a processor’s behaviour should be when one of the
headers is missing.

If you need to fit arbitrarily complex data into a header, Structured
Headers is probably a poor fit for your use case.

B.3. What should generic Structured Headers implementations expose?

A generic implementation should expose the top-level parse
(Section 4.2) and serialise (Section 4.1) 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; see
https://github.com/httpwg/structured-header-tests [7].

Implementers should note that dictionaries and parameters are order-
preserving maps. Some headers 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.

Appendix C. Changes

_RFC Editor: Please remove this section before publication._

C.1. Since draft-ietf-httpbis-header-structure-09

- Changed Boolean from T/F to 1/0 (#784).
- Parameters are now ordered maps (#765).
C.2. Since draft-ietf-httpbis-header-structure-08
   o Clamp integers to 15 digits (#737).
   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.3. 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.4. 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.5. 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.
  o Remove identifiers from item.
  o Remove most limits on sizes.
  o Refine number parsing.
C.7. Since draft-ietf-httpbis-header-structure-03
  o Strengthen language around failure handling.
C.8. 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.
Internet-Draft Structured Headers for HTTP April 2019

- Throw an error on trailing garbage.

- Replaced with draft-nottingham-structured-headers.

C.10. Since draft-ietf-httpbis-header-structure-00
- Added signed 64bit integer type.
- Drop UTF8, and settle on BCP137 ::EmbeddedUnicodeChar for h1-unicode-string.
- Change h1_blob delimiter to "::" since "'" is valid t_char

Authors’ Addresses

Mark Nottingham
Fastly

Email: mnot@mnot.net
URI: https://www.mnot.net/

Poul-Henning Kamp
The Varnish Cache Project

Email: phk@varnish-cache.org
Secondary Certificate Authentication in HTTP/2

draft-ietf-httpbis-http2-secondary-certs-04

Abstract

A use of TLS Exported Authenticators is described which enables
HTTP/2 clients and servers to offer additional certificate-based
credentials after the connection is established. The means by which
these credentials are used with requests is defined.

Note to Readers

Discussion of this draft takes place on the HTTP working group
mailing list (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

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 October 27, 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 (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. Server Certificate Authentication ......................... 3
   1.2. Client Certificate Authentication ........................ 4
       1.2.1. HTTP/1.1 Using TLS 1.2 and Earlier ...................... 5
       1.2.2. HTTP/1.1 Using TLS 1.3 ................................. 6
       1.2.3. HTTP/2 ................................................. 6
   1.3. HTTP-Layer Certificate Authentication ...................... 7
   1.4. Terminology ............................................. 8
2. Discovering Additional Certificates at the HTTP/2 Layer ........ 8
   2.1. Indicating Support for HTTP-Layer Certificate
       Authentication ............................................. 8
   2.2. Making Certificates or Requests Available ................. 9
   2.3. Requiring Certificate Authentication ...................... 10
       2.3.1. Requiring Additional Server Certificates ............... 10
       2.3.2. Requiring Additional Client Certificates .......... 11
3. Certificates Frames for HTTP/2 ................................ 12
   3.1. The CERTIFICATE_NEEDED Frame ............................ 13
   3.2. The USE_CERTIFICATE Frame ................................. 14
   3.3. The CERTIFICATE_REQUEST Frame ............................. 15
       3.3.1. Exported Authenticator Request Characteristics ...... 16
   3.4. The CERTIFICATE Frame ..................................... 16
       3.4.1. Exported Authenticator Characteristics ................. 18
4. Indicating Failures During HTTP-Layer Certificate
   Authentication ............................................. 18
5. Required Domain Certificate Extension .......................... 19
6. Security Considerations ..................................... 20
   6.1. Impersonation ............................................. 20
   6.2. Fingerprinting ............................................ 21
   6.3. Denial of Service ......................................... 21
   6.4. Persistence of Service ...................................... 21
   6.5. Confusion About State ..................................... 22
HTTP clients need to know that the content they receive on a connection comes from the origin that they intended to retrieve it from. The traditional form of server authentication in HTTP has been in the form of a single X.509 certificate provided during the TLS ([RFC5246], [RFC8446]) handshake.

Many existing HTTP [RFC7230] servers also have authentication requirements for the resources they serve. Of the bountiful authentication options available for authenticating HTTP requests, client certificates present a unique challenge for resource-specific authentication requirements because of the interaction with the underlying TLS layer.

TLS 1.2 [RFC5246] supports one server and one client certificate on a connection. These certificates may contain multiple identities, but only one certificate may be provided.

Many HTTP servers host content from several origins. HTTP/2 permits clients to reuse an existing HTTP connection to a server provided that the secondary origin is also in the certificate provided during the TLS handshake. In many cases, servers choose to maintain separate certificates for different origins but still desire the benefits of a shared HTTP connection.

1.1. Server Certificate Authentication

Section 9.1.1 of [RFC7540] describes how connections may be used to make requests from multiple origins as long as the server is authoritative for both. A server is considered authoritative for an
origin if DNS resolves the origin to the IP address of the server and (for TLS) if the certificate presented by the server contains the origin in the Subject Alternative Names field.

[RFC7838] enables a step of abstraction from the DNS resolution. If both hosts have provided an Alternative Service at hostnames which resolve to the IP address of the server, they are considered authoritative just as if DNS resolved the origin itself to that address. However, the server’s one TLS certificate is still required to contain the name of each origin in question.

[RFC8336] relaxes the requirement to perform the DNS lookup if already connected to a server with an appropriate certificate which claims support for a particular origin.

Servers which host many origins often would prefer to have separate certificates for some sets of origins. This may be for ease of certificate management (the ability to separately revoke or renew them), due to different sources of certificates (a CDN acting on behalf of multiple origins), or other factors which might drive this administrative decision. Clients connecting to such origins cannot currently reuse connections, even if both client and server would prefer to do so.

Because the TLS SNI extension is exchanged in the clear, clients might also prefer to retrieve certificates inside the encrypted context. When this information is sensitive, it might be advantageous to request a general-purpose certificate or anonymous ciphersuite at the TLS layer, while acquiring the "real" certificate in HTTP after the connection is established.

1.2. Client Certificate Authentication

For servers that wish to use client certificates to authenticate users, they might request client authentication during or immediately after the TLS handshake. However, if not all users or resources need certificate-based authentication, a request for a certificate has the unfortunate consequence of triggering the client to seek a certificate, possibly requiring user interaction, network traffic, or other time-consuming activities. During this time, the connection is stalled in many implementations. Such a request can result in a poor experience, particularly when sent to a client that does not expect the request.

The TLS 1.3 CertificateRequest can be used by servers to give clients hints about which certificate to offer. Servers that rely on certificate-based authentication might request different certificates for different resources. Such a server cannot use contextual
information about the resource to construct an appropriate TLS CertificateRequest message during the initial handshake.

Consequently, client certificates are requested at connection establishment time only in cases where all clients are expected or required to have a single certificate that is used for all resources. Many other uses for client certificates are reactive, that is, certificates are requested in response to the client making a request.

1.2.1. HTTP/1.1 Using TLS 1.2 and Earlier

In HTTP/1.1, a server that relies on client authentication for a subset of users or resources does not request a certificate when the connection is established. Instead, it only requests a client certificate when a request is made to a resource that requires a certificate. TLS 1.2 [RFC5246] accommodates this by permitting the server to request a new TLS handshake, in which the server will request the client’s certificate.

Figure 1 shows the server initiating a TLS-layer renegotiation in response to receiving an HTTP/1.1 request to a protected resource.

```
Client                                      Server
-- (HTTP) GET /protected -------------------> *1
<---------------------- (TLS) HelloRequest -- *2
-- (TLS) ClientHello ----------------------->
<------------------ (TLS) ServerHello, ... --
<---------------- (TLS) CertificateRequest -- *3
-- (TLS) ..., Certificate ------------------> *4
-- (TLS) Finished -------------------------->
<-------------------------- (TLS) Finished --
<--------------------------- (HTTP) 200 OK -- *5
```

**Figure 1: HTTP/1.1 reactive certificate authentication with TLS 1.2**

In this example, the server receives a request for a protected resource (at *1 on Figure 1). Upon performing an authorization check, the server determines that the request requires authentication using a client certificate and that no such certificate has been provided.

The server initiates TLS renegotiation by sending a TLS HelloRequest (at *2). The client then initiates a TLS handshake. Note that some TLS messages are elided from the figure for the sake of brevity.

The critical messages for this example are the server requesting a certificate with a TLS CertificateRequest (*3); this request might
use information about the request or resource. The client then provides a certificate and proof of possession of the private key in Certificate and CertificateVerify messages (*4).

When the handshake completes, the server performs any authorization checks a second time. With the client certificate available, it then authorizes the request and provides a response (*5).

1.2.2. HTTP/1.1 Using TLS 1.3

TLS 1.3 [RFC8446] introduces a new client authentication mechanism that allows for clients to authenticate after the handshake has been completed. For the purposes of authenticating an HTTP request, this is functionally equivalent to renegotiation. Figure 2 shows the simpler exchange this enables.

Client
-- (HTTP) GET /protected ------------------->

Server
<!-----------------(TLS) CertificateRequest --

-- (TLS) Certificate, CertificateVerify, Finished ----------------------->

<!---------------------------(HTTP) 200 OK --

Figure 2: HTTP/1.1 reactive certificate authentication with TLS 1.3

TLS 1.3 does not support renegotiation, instead supporting direct client authentication. In contrast to the TLS 1.2 example, in TLS 1.3, a server can simply request a certificate.

1.2.3. HTTP/2

An important part of the HTTP/1.1 exchange is that the client is able to easily identify the request that caused the TLS renegotiation. The client is able to assume that the next unanswered request on the connection is responsible. The HTTP stack in the client is then able to direct the certificate request to the application or component that initiated that request. This ensures that the application has the right contextual information for processing the request.

In HTTP/2, a client can have multiple outstanding requests. Without some sort of correlation information, a client is unable to identify which request caused the server to request a certificate.

Thus, the minimum necessary mechanism to support reactive certificate authentication in HTTP/2 is an identifier that can be used to correlate an HTTP request with a request for a certificate. Since streams are used for individual requests, correlation with a stream is sufficient.
[RFC7540] prohibits renegotiation after any application data has been sent. This completely blocks reactive certificate authentication in HTTP/2 using TLS 1.2. If this restriction were relaxed by an extension or update to HTTP/2, such an identifier could be added to TLS 1.2 by means of an extension to TLS. Unfortunately, many TLS 1.2 implementations do not permit application data to continue during a renegotiation. This is problematic for a multiplexed protocol like HTTP/2.

1.3. HTTP-Layer Certificate Authentication

This draft defines HTTP/2 frames to carry the relevant certificate messages, enabling certificate-based authentication of both clients and servers independent of TLS version. This mechanism can be implemented at the HTTP layer without breaking the existing interface between HTTP and applications above it.

This could be done in a naive manner by replicating the TLS messages as HTTP/2 frames on each stream. However, this would create needless redundancy between streams and require frequent expensive signing operations. Instead, TLS Exported Authenticators [I-D.ietf-tls-exported-authenticator] are exchanged on stream zero and other frames incorporate them to particular requests by reference as needed.

TLS Exported Authenticators are structured messages that can be exported by either party of a TLS connection and validated by the other party. Given an established TLS connection, a request can be constructed which describes the desired certificate and an authenticator message can be constructed proving possession of a certificate and a corresponding private key. Both requests and authenticators can be generated by either the client or the server. Exported Authenticators use the message structures from Sections 4.3.2 and 4.4 of [RFC8446], but different parameters.

Each Authenticator is computed using a Handshake Context and Finished MAC Key derived from the TLS session. The Handshake Context is identical for both parties of the TLS connection, while the Finished MAC Key is dependent on whether the Authenticator is created by the client or the server.

Successfully verified Authenticators result in certificate chains, with verified possession of the corresponding private key, which can be supplied into a collection of available certificates. Likewise, descriptions of desired certificates can be supplied into these collections.
Section 2 describes how the feature is employed, defining means to
detect support in peers (Section 2.1), make certificates and requests
available (Section 2.2), and indicate when streams are blocked
waiting on an appropriate certificate (Section 2.3). Section 3
defines the required frame types, which parallel the TLS 1.3 message
exchange. Finally, Section 4 defines new error types which can be
used to notify peers when the exchange has not been successful.

1.4. 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.

2. Discovering Additional Certificates at the HTTP/2 Layer

A certificate chain with proof of possession of the private key
corresponding to the end-entity certificate is sent as a sequence of
"CERTIFICATE" frames (see Section 3.4) on stream zero. Once the
holder of a certificate has sent the chain and proof, this
certificate chain is cached by the recipient and available for future
use. Clients can proactively indicate the certificate they intend to
use on each request using an unsolicited "USE_CERTIFICATE" frame, if
desired. The previously-supplied certificates are available for
reference without having to resend them.

Otherwise, the server uses a "CERTIFICATE_REQUEST" frame to describe
a class of certificates on stream zero, then uses
"CERTIFICATE_NEEDED" frames to associate these with individual
requests. The client responds with a "USE_CERTIFICATE" frame
indicating the certificate which should be used to satisfy the
request.

Data sent by each peer is correlated by the ID given in each frame.
This ID is unrelated to values used by the other peer, even if each
uses the same ID in certain cases. "USE_CERTIFICATE" frames indicate
whether they are sent proactively or are in response to a
"CERTIFICATE_NEEDED" frame.

2.1. Indicating Support for HTTP-Layer Certificate Authentication

Clients and servers that will accept requests for HTTP-layer
certificate authentication indicate this using the HTTP/2
"SETTINGS_HTTP_CERT_AUTH" (0xSETTING-TBD) setting.
The initial value for the "SETTINGS_HTTP_CERT_AUTH" setting is 0, indicating that the peer does not support HTTP-layer certificate authentication. If a peer does support HTTP-layer certificate authentication, the value is non-zero.

In order to ensure that the TLS connection is direct to the server, rather than via a TLS-terminating proxy, each side will separately compute and confirm the value of this setting. The setting is derived from a TLS exporter (see Section 7.5 of [RFC8446] and [RFC5705] for more details on exporters). Clients MUST NOT use an early exporter during their 0-RTT flight, but MUST send an updated SETTINGS frame using a regular exporter after the TLS handshake completes.

The exporter is constructed with the following input:

- Label:
  * "EXPORTER HTTP CERTIFICATE client" for clients
  * "EXPORTER HTTP CERTIFICATE server" for servers
- Context: Empty
- Length: Four bytes

The resulting exporter is converted to a setting value as:

\[(\text{Exporter} \& 0x3fffffff) | 0x80000000\]

That is, the most significant bit will always be set, regardless of the value of the exporter. Each endpoint will compute the expected value from their peer. If the setting is not received, or if the value received is not the expected value, the frames defined in this document SHOULD NOT be sent.

2.2. Making Certificates or Requests Available

When both peers have advertised support for HTTP-layer certificates as in Section 2.1, either party can supply additional certificates into the connection at any time. This means that clients or servers which predict a certificate will be required could supply the certificate before being asked. These certificates are available for reference by future "USE_CERTIFICATE" frames.

Certificates supplied by servers can be considered by clients without further action by the server. A server SHOULD NOT send certificates which do not cover origins which it is prepared to service on the
current connection, but MAY use the ORIGIN frame [RFC8336] to indicate that not all covered origins will be served.

Client                                      Server
<------------------- (stream 0) CERTIFICATE --
...-- (stream N) GET /from-new-origin -------->
<------------------- (stream N) 200 OK --

Figure 3: Proactive server authentication

Client                                      Server
-- (stream 0) CERTIFICATE ------------------>
-- (stream 0) USE_CERTIFICATE (S=1) -------->
-- (stream 0) USE_CERTIFICATE (S=3) -------->
-- (streams 1,3) GET /protected ----------->
<------------------- (streams 1,3) 200 OK --

Figure 4: Proactive client authentication

Likewise, either party can supply a "CERTIFICATE_REQUEST" that outlines parameters of a certificate they might request in the future. Upon receipt of a "CERTIFICATE_REQUEST", endpoints SHOULD provide a corresponding certificate in anticipation of a request shortly being blocked. Clients MAY wait for a "CERTIFICATE_NEEDED" frame to assist in associating the certificate request with a particular HTTP transaction.

2.3. Requiring Certificate Authentication

2.3.1. Requiring Additional Server Certificates

As defined in [RFC7540], when a client finds that an https:// origin (or Alternative Service [RFC7838]) to which it needs to make a request has the same IP address as a server to which it is already connected, it MAY check whether the TLS certificate provided contains the new origin as well, and if so, reuse the connection.

If the TLS certificate does not contain the new origin, but the server has claimed support for that origin (with an ORIGIN frame, see [RFC8336]) and advertised support for HTTP-layer certificates (see Section 2.1), the client MAY send a "CERTIFICATE_REQUEST" frame describing the desired origin. The client then sends a "CERTIFICATE_NEEDED" frame for stream zero referencing the request, indicating that the connection cannot be used for that origin until the certificate is provided.
If the server does not have the desired certificate, it MUST send an Empty Authenticator, as described in Section 5 of [I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE_CERTIFICATE" frame for stream zero which references the Empty Authenticator. In this case, or if the server has not advertised support for HTTP-layer certificates, the client MUST NOT send any requests for resources in that origin on the current connection.

Client                                      Server
<----------------------- (stream 0) ORIGIN --
-- (stream 0) CERTIFICATE_REQUEST -------->
-- (stream 0) CERTIFICATE_NEEDED (S=0) ---->
<------------------- (stream 0) CERTIFICATE --
<-------- (stream 0) USE_CERTIFICATE (S=0) --
-- (stream N) GET /from-new-origin -------->
<---------------- (stream N) 200 OK --

Figure 5: Client-requested certificate

If a client receives a "PUSH_PROMISE" referencing an origin for which it has not yet received the server’s certificate, this is a fatal connection error (see section 8.2 of [RFC7540]). To avoid this, servers MUST supply the associated certificates before pushing resources from a different origin.

2.3.2. Requiring Additional Client Certificates

Likewise, the server sends a "CERTIFICATE_NEEDED" frame for each stream where certificate authentication is required. The client answers with a "USE_CERTIFICATE" frame indicating the certificate to use on that stream. If the request parameters or the responding certificate are not already available, they will need to be sent as described in Section 2.2 as part of this exchange.

Client                                      Server
<---------- (stream 0) CERTIFICATE_REQUEST --
...  
-- (stream N) GET /protected ---------------
<----- (stream 0) CERTIFICATE_NEEDED (S=N) --
-- (stream 0) CERTIFICATE ------------------
-- (stream 0) USE_CERTIFICATE (S=N) ------>
<---------------- (stream N) 200 OK --

Figure 6: Reactive certificate authentication

If the client does not have the desired certificate, it instead sends an Empty Authenticator, as described in Section 5 of
[I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE_CERTIFICATE" frame which references the Empty Authenticator. In this case, or if the client has not advertised support for HTTP-layer certificates, the server processes the request based solely on the certificate provided during the TLS handshake, if any. This might result in an error response via HTTP, such as a status code 403 (Not Authorized).

3. Certificates Frames for HTTP/2

The "CERTIFICATE_REQUEST" and "CERTIFICATE_NEEDED" frames are correlated by their "Request-ID" field. Subsequent "CERTIFICATE_NEEDED" frames with the same "Request-ID" value MAY be sent for other streams where the sender is expecting a certificate with the same parameters.

The "CERTIFICATE", and "USE_CERTIFICATE" frames are correlated by their "Cert-ID" field. Subsequent "USE_CERTIFICATE" frames with the same "Cert-ID" MAY be sent in response to other "CERTIFICATE_NEEDED" frames and refer to the same certificate.

"CERTIFICATE_NEEDED" and "USE_CERTIFICATE" frames are correlated by the Stream ID they reference. Unsolicited "USE_CERTIFICATE" frames are not responses to "CERTIFICATE_NEEDED" frames; otherwise, each "USE_CERTIFICATE" frame for a stream is considered to respond to a "CERTIFICATE_NEEDED" frame for the same stream in sequence.

```
+---------+           +---------+
| REQUEST |           | CERT    |
+---------+           +---------+
    +--------+           +--------+
    | Request-ID | Cert-ID |
    +--------+           +--------+
                v           v
            +--------+ Stream ID +--------+
            | NEEDED |       -------------> USE |
            +--------+           +--------+
```

Figure 7: Frame correlation

"Request-ID" and "Cert-ID" are independent and sender-local. The use of the same value by the other peer or in the other context does not imply any correlation between these frames. These values MUST be unique per sender for each space over the lifetime of the connection.
3.1. The CERTIFICATE_NEEDED Frame

The "CERTIFICATE_NEEDED" frame (0xFRAME-TBD1) is sent on stream zero to indicate that the HTTP request on the indicated stream is blocked pending certificate authentication. The frame includes stream ID and a request identifier which can be used to correlate the stream with a previous "CERTIFICATE_REQUEST" frame sent on stream zero. The "CERTIFICATE_REQUEST" describes the certificate the sender requires to make progress on the stream in question.

```
0                   1                   2                   3
+---------------------------------------------------------------+
|R| Stream ID (31) |
+-------------------------------+-------------------------------+
    Request-ID (16) |
                +-------------------------------+
```

Figure 8: CERTIFICATE_NEEDED frame payload

The "CERTIFICATE_NEEDED" frame contains 6 octets. The first four octets indicate the Stream ID of the affected stream. The following two octets are the authentication request identifier, "Request-ID". A peer that receives a "CERTIFICATE_NEEDED" of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical request identifiers refer to the same "CERTIFICATE_REQUEST".

A server MAY send multiple "CERTIFICATE_NEEDED" frames for the same stream. If a server requires that a client provide multiple certificates before authorizing a single request, each required certificate MUST be indicated with a separate "CERTIFICATE_NEEDED" frame, each of which MUST have a different request identifier (referencing different "CERTIFICATE_REQUEST" frames describing each required certificate). To reduce the risk of client confusion, servers SHOULD NOT have multiple outstanding "CERTIFICATE_NEEDED" frames for the same stream at any given time.

Clients MUST only send multiple "CERTIFICATE_NEEDED" frames for stream zero. Multiple "CERTIFICATE_NEEDED" frames on any other stream MUST be considered a stream error of type "PROTOCOL_ERROR".

The "CERTIFICATE_NEEDED" frame MUST NOT be sent to a peer which has not advertised support for HTTP-layer certificate authentication.

The "CERTIFICATE_NEEDED" frame MUST NOT reference a stream in the "half-closed (local)" or "closed" states [RFC7540]. A client that receives a "CERTIFICATE_NEEDED" frame for a stream which is not in a
valid state SHOULD treat this as a stream error of type
"PROTOCOL_ERROR".

3.2. The USE_CERTIFICATE Frame

The "USE_CERTIFICATE" frame (0xFRAME-TBD4) is sent on stream zero to
indicate which certificate is being used on a particular request
stream.

The "USE_CERTIFICATE" frame defines a single flag:

UNSOLICITED (0x01): Indicates that no "CERTIFICATE_NEEDED" frame has
yet been received for this stream.

The payload of the "USE_CERTIFICATE" frame is as follows:

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------------------------------------------------------+
|R|                        Stream ID (31)                       |
+-------------------------------+-------------------------------+
|         [Cert-ID (16)]        |
+-------------------------------+
```

Figure 9: USE_CERTIFICATE frame payload

The first four octets indicate the Stream ID of the affected stream.
The following two octets, if present, contain the two-octet "Cert-ID"
of the certificate the sender wishes to use. This MUST be the ID of
a certificate for which proof of possession has been presented in a
"CERTIFICATE" frame. Recipients of a "USE_CERTIFICATE" frame of any
other length MUST treat this as a stream error of type
"PROTOCOL_ERROR". Frames with identical certificate identifiers
refer to the same certificate chain.

A "USE_CERTIFICATE" frame which omits the Cert-ID refers to the
certificate provided at the TLS layer, if any. If no certificate was
provided at the TLS layer, the stream should be processed with no
authentication, likely returning an authentication-related error at
the HTTP level (e.g. 403) for servers or routing the request to a new
connection for clients.

The "UNSOLICITED" flag MAY be set by clients on the first
"USE_CERTIFICATE" frame referring to a given stream. This permits a
client to proactively indicate which certificate should be used when
processing a new request. When such an unsolicited indication refers
to a request that has not yet been received, servers SHOULD cache the
indication briefly in anticipation of the request.
Receipt of more than one unsolicited "USE_CERTIFICATE" frames or an unsolicited "USE_CERTIFICATE" frame which is not the first in reference to a given stream MUST be treated as a stream error of type "CERTIFICATE_OVERUSED".

Each "USE_CERTIFICATE" frame which is not marked as unsolicited is considered to respond in order to the "CERTIFICATE_NEEDED" frames for the same stream. If a "USE_CERTIFICATE" frame is received for which a "CERTIFICATE_NEEDED" frame has not been sent, this MUST be treated as a stream error of type "CERTIFICATE_OVERUSED".

Receipt of a "USE_CERTIFICATE" frame with an unknown "Cert-ID" MUST result in a stream error of type "PROTOCOL_ERROR".

The referenced certificate chain needs to conform to the requirements expressed in the "CERTIFICATE_REQUEST" to the best of the sender’s ability, or the recipient is likely to reject it as unsuitable despite properly validating the authenticator. If the recipient considers the certificate unsuitable, it MAY at its discretion either return an error at the HTTP semantic layer, or respond with a stream error [RFC7540] on any stream where the certificate is used. Section 4 defines certificate-related error codes which might be applicable.

3.3. The CERTIFICATE_REQUEST Frame

The "CERTIFICATE_REQUEST" frame (id=0xFRAME-TBD2) provides an exported authenticator request message from the TLS layer that specifies a desired certificate. This describes the certificate the sender wishes to have presented.

The "CERTIFICATE_REQUEST" frame SHOULD NOT be sent to a peer which has not advertised support for HTTP-layer certificate authentication.

The "CERTIFICATE_REQUEST" frame MUST be sent on stream zero. A "CERTIFICATE_REQUEST" frame received on any other stream MUST be rejected with a stream error of type "PROTOCOL_ERROR".

```
+-------------------------------+-------------------------------+
|        Request-ID (16)        |          Request (?)        ...
|-------------------------------+-------------------------------+
```

Figure 10: CERTIFICATE_REQUEST frame payload

The frame contains the following fields:
Request-ID: "Request-ID" is a 16-bit opaque identifier used to correlate subsequent certificate-related frames with this request. The identifier MUST be unique in the session for the sender.

Request: An exported authenticator request, generated using the "request" API described in [I-D.ietf-tls-exported-authenticator]. See Section 3.4.1 for more details on the input to this API.

3.3.1. Exported Authenticator Request Characteristics

The Exported Authenticator "request" API defined in [I-D.ietf-tls-exported-authenticator] takes as input a set of desired certificate characteristics and a "certificate_request_context", which needs to be unpredictable. When generating exported authenticators for use with this extension, the "certificate_request_context" MUST contain both the two-octet Request-ID as well as at least 96 bits of additional entropy.

Upon receipt of a "CERTIFICATE_REQUEST" frame, the recipient MUST verify that the first two octets of the authenticator’s "certificate_request_context" matches the Request-ID presented in the frame.

The TLS library on the authenticating peer will provide mechanisms to select an appropriate certificate to respond to the transported request. TLS libraries on servers MUST be able to recognize the "server_name" extension ([RFC6066]) at a minimum. Clients MUST always specify the desired origin using this extension, though other extensions MAY also be included.

3.4. The CERTIFICATE Frame

The "CERTIFICATE" frame (id=0xFRAME-TBD3) provides an exported authenticator message from the TLS layer that provides a chain of certificates, associated extensions and proves possession of the private key corresponding to the end-entity certificate.

The "CERTIFICATE" frame defines two flags:

TO_BE_CONTINUED (0x01): Indicates that the exported authenticator spans more than one frame.

UNSOLICITED (0x02): Indicates that the exported authenticator does not contain a Request-ID.
The frame contains the following fields:

Cert-ID: "Cert-ID" is a 16-bit opaque identifier used to correlate other certificate-related frames with this exported authenticator fragment.

Request-ID: "Request-ID" is an optional 16-bit opaque identifier used to correlate this exported authenticator with the request which triggered it, if any. This field is present only if the "UNSOLICITED" flag is not set.

Authenticator Fragment: A portion of the opaque data returned from the TLS connection exported authenticator "authenticate" API. See Section 3.4.1 for more details on the input to this API.

An exported authenticator is transported in zero or more "CERTIFICATE" frames with the "TO_BE_CONTINUED" flag set, followed by one "CERTIFICATE" frame with the "TO_BE_CONTINUED" flag unset. Each of these frames contains the same "Cert-ID" field, permitting them to be associated with each other. Receipt of any "CERTIFICATE" frame with the same "Cert-ID" following the receipt of a "CERTIFICATE" frame with "TO_BE_CONTINUED" unset MUST be treated as a connection error of type "PROTOCOL_ERROR".

If the "UNSOLICITED" flag is not set, the "CERTIFICATE" frame also contains a Request-ID indicating the certificate request which caused this exported authenticator to be generated. The value of this flag and the contents of the Request-ID field MUST NOT differ between frames with the same Cert-ID.

Upon receiving a complete series of "CERTIFICATE" frames, the receiver may validate the Exported Authenticator value by using the exported authenticator API. This returns either an error indicating that the message was invalid, or the certificate chain and extensions used to create the message.
The "CERTIFICATE" frame MUST be sent on stream zero. A "CERTIFICATE" frame received on any other stream MUST be rejected with a stream error of type "PROTOCOL_ERROR".

3.4.1. Exported Authenticator Characteristics

The Exported Authenticator API defined in [I-D.ietf-tls-exported-authenticator] takes as input a request, a set of certificates, and supporting information about the certificate (OCSP, SCT, etc.). The result is an opaque token which is used when generating the "CERTIFICATE" frame.

Upon receipt of a "CERTIFICATE" frame, an endpoint MUST perform the following steps to validate the token it contains:

- Verify that either the "UNSOLICITED" flag is set (clients only) or that the Request-ID field contains the Request-ID of a previously-sent "CERTIFICATE_REQUEST" frame.

- Using the "get context" API, retrieve the "certificate_request_context" used to generate the authenticator, if any. Verify that the "certificate_request_context" begins with the supplied Request-ID, if any.

- Use the "validate" API to confirm the validity of the authenticator with regard to the generated request (if any).

Once the authenticator is accepted, the endpoint can perform any other checks for the acceptability of the certificate itself. Clients MUST NOT accept any end-entity certificate from an exported authenticator which does not contain the Required Domain extension; see Section 5 and Section 6.1.

4. Indicating Failures During HTTP-Layer Certificate Authentication

Because this draft permits certificates to be exchanged at the HTTP framing layer instead of the TLS layer, several certificate-related errors which are defined at the TLS layer might now occur at the HTTP framing layer. In this section, those errors are restated and added to the HTTP/2 error code registry.

BAD_CERTIFICATE (0xERROR-TBD1): A certificate was corrupt, contained signatures that did not verify correctly, etc.

UNSUPPORTED_CERTIFICATE (0xERROR-TBD2): A certificate was of an unsupported type or did not contain required extensions
CERTIFICATE_REVOKED (0xERROR-TBD3): A certificate was revoked by its signer

CERTIFICATE_EXPIRED (0xERROR-TBD4): A certificate has expired or is not currently valid

CERTIFICATE_GENERAL (0xERROR-TBD5): Any other certificate-related error

CERTIFICATE_OVERUSED (0xERROR-TBD6): More certificates were used on a request than were requested

As described in [RFC7540], implementations MAY choose to treat a stream error as a connection error at any time. Of particular note, a stream error cannot occur on stream 0, which means that implementations cannot send non-session errors in response to "CERTIFICATE_REQUEST", and "CERTIFICATE" frames. Implementations which do not wish to terminate the connection MAY either send relevant errors on any stream which references the failing certificate in question or process the requests as unauthenticated and provide error information at the HTTP semantic layer.

5. Required Domain Certificate Extension

The Required Domain extension allows certificates to limit their use with Secondary Certificate Authentication. A client MUST verify that the server has proven ownership of the indicated identity before accepting the limited certificate over Secondary Certificate Authentication.

The identity in this extension is a restriction asserted by the requester of the certificate and is not verified by the CA. Conforming CAs SHOULD mark the requiredDomain extension as non-critical. Conforming CAs MUST require the presence of a CAA record [RFC6844] prior to issuing a certificate with this extension. Because a Required Domain value of ".*" has a much higher risk of reuse if compromised, conforming Certificate Authorities are encouraged to require more extensive verification prior to issuing such a certificate.

The required domain is represented as a GeneralName, as specified in Section 4.2.1.6 of [RFC5280]. Unlike the subject field, conforming CAs MUST NOT issue certificates with a requiredDomain extension containing empty GeneralName fields. Clients that encounter such a certificate when processing a certification path MUST consider the certificate invalid.
The wildcard character "_" MAY be used to represent that any previously authenticated identity is acceptable. This character MUST be the entirety of the name if used and MUST have a type of "dNSName". (That is, "_" is acceptable, but "_.com" and "w_.example.com" are not).

\[
\text{id-ce-requiredDomain OBJECT IDENTIFIER ::= \{ id-ce \text{TBD1} \}}
\]

\[
\text{RequiredDomain ::= GeneralName}
\]

6. Security Considerations

This mechanism defines an alternate way to obtain server and client certificates other than in the initial TLS handshake. While the signature of exported authenticator values is expected to be equally secure, it is important to recognize that a vulnerability in this code path is at least equal to a vulnerability in the TLS handshake.

6.1. Impersonation

This mechanism could increase the impact of a key compromise. Rather than needing to subvert DNS or IP routing in order to use a compromised certificate, a malicious server now only needs a client to connect to _some_ HTTPS site under its control in order to present the compromised certificate. As recommended in [RFC8336], clients opting not to consult DNS ought to employ some alternative means to increase confidence that the certificate is legitimate.

One such means is the Required Domain certificate extension defined in {extension}. Clients MUST require that server certificates presented via this mechanism contain the Required Domain extension and require that a certificate previously accepted on the connection (including the certificate presented in TLS) lists the Required Domain in the Subject field or the Subject Alternative Name extension.

As noted in the Security Considerations of [I-D.ietf-tls-exported-authenticator], it is difficult to formally prove that an endpoint is jointly authoritative over multiple certificates, rather than individually authoritative on each certificate. As a result, clients MUST NOT assume that because one origin was previously colocated with another, those origins will be reachable via the same endpoints in the future. Clients MUST NOT consider previous secondary certificates to be validated after TLS session resumption. However, clients MAY proactively query for previously-presented secondary certificates.
6.2. Fingerprinting

This draft defines a mechanism which could be used to probe servers for origins they support, but opens no new attack versus making repeat TLS connections with different SNI values. Servers SHOULD impose similar denial-of-service mitigations (e.g. request rate limits) to "CERTIFICATE_REQUEST" frames as to new TLS connections.

While the extensions in the "CERTIFICATE_REQUEST" frame permit the sender to enumerate the acceptable Certificate Authorities for the requested certificate, it might not be prudent (either for security or data consumption) to include the full list of trusted Certificate Authorities in every request. Senders, particularly clients, SHOULD send only the extensions that narrowly specify which certificates would be acceptable.

6.3. Denial of Service

Failure to provide a certificate on a stream after receiving "CERTIFICATE_NEEDED" blocks processing, and SHOULD be subject to standard timeouts used to guard against unresponsive peers.

Validating a multitude of signatures can be computationally expensive, while generating an invalid signature is computationally cheap. Implementations will require checks for attacks from this direction. Invalid exported authenticators SHOULD be treated as a session error, to avoid further attacks from the peer, though an implementation MAY instead disable HTTP-layer certificates for the current connection instead.

6.4. Persistence of Service

CNAME records in the DNS are frequently used to delegate authority for an origin to a third-party provider. This delegation can be changed without notice, even to the third-party provider, simply by modifying the CNAME record in question.

After the owner of the domain has redirected traffic elsewhere by changing the CNAME, new connections will not arrive for that origin, but connections which are properly directed to this provider for other origins would continue to claim control of this origin (via ORIGIN frame and Secondary Certificates). This is proper behavior based on the third-party provider’s configuration, but would likely not be what is intended by the owner of the origin.

This is not an issue which can be mitigated by the protocol, but something about which third-party providers SHOULD educate their customers before using the features described in this document.
6.5. Confusion About State

Implementations need to be aware of the potential for confusion about the state of a connection. The presence or absence of a validated certificate can change during the processing of a request, potentially multiple times, as "USE_CERTIFICATE" frames are received. A server that uses certificate authentication needs to be prepared to reevaluate the authorization state of a request as the set of certificates changes.

Client implementations need to carefully consider the impact of setting the "AUTOMATIC_USE" flag. This flag is a performance optimization, permitting the client to avoid a round-trip on each request where the server checks for certificate authentication. However, once this flag has been sent, the client has zero knowledge about whether the server will use the referenced cert for any future request, or even for an existing request which has not yet completed. Clients MUST NOT set this flag on any certificate which is not appropriate for currently-in-flight requests, and MUST NOT make any future requests on the same connection which they are not willing to have associated with the provided certificate.

7. IANA Considerations

This draft adds entries in three registries.

The HTTP/2 "SETTINGS_HTTP_CERT_AUTH" setting is registered in Section 7.1. Four frame types are registered in Section 7.2. Six error codes are registered in Section 7.3.

7.1. HTTP/2 SETTINGS_HTTP_CERT_AUTH Setting

The SETTINGS_HTTP_CERT_AUTH setting is registered in the "HTTP/2 Settings" registry established in [RFC7540].

Name: SETTINGS_HTTP_CERT_AUTH

Code: 0xSETTING-TBD

Initial Value: 0

Specification: This document.

7.2. New HTTP/2 Frames

Four new frame types are registered in the "HTTP/2 Frame Types" registry established in [RFC7540]. The entries in the following table are registered by this document.
### 7.3. New HTTP/2 Error Codes

Six new error codes are registered in the "HTTP/2 Error Code" registry established in [RFC7540]. The entries in the following table are registered by this document.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_CERTIFICATE</td>
<td>0xERROR-TBD1</td>
<td>Section 4</td>
</tr>
<tr>
<td>UNSUPPORTED_CERTIFICATE</td>
<td>0xERROR-TBD2</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_REVOKED</td>
<td>0xERROR-TBD3</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_EXPIRED</td>
<td>0xERROR-TBD4</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_GENERAL</td>
<td>0xERROR-TBD5</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_OVERUSED</td>
<td>0xERROR-TBD6</td>
<td>Section 4</td>
</tr>
</tbody>
</table>

### 8. References

#### 8.1. Normative References

[I-D.ietf-tls-exported-authenticator]

Informative References


8.2. Informative References


8.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/


Appendix A. Change Log

*RFC Editor’s Note:* Please remove this section prior to publication of a final version of this document.

A.1. Since draft-ietf-httpbis-http2-secondary-certs-03:

- "CERTIFICATE_REQUEST" frames contain the Request-ID, which MUST be checked against the "certificate_request_context" of the Exported Authenticator Request.
- "CERTIFICATE" frames contain the Request-ID to which they respond, unless the UNSOLICITED flag is set.
- The Required Domain extension is defined for certificates, which must be present for certificates presented by servers.

A.2. Since draft-ietf-httpbis-http2-secondary-certs-02:

Editorial updates only.

A.3. Since draft-ietf-httpbis-http2-secondary-certs-01:

- Clients can send "CERTIFICATE_NEEDED" for stream 0 rather than speculatively reserving a stream for an origin.
- Use SETTINGS to disable when a TLS-terminating proxy is present (#617,#651)

A.4. Since draft-ietf-httpbis-http2-secondary-certs-00:

- All frames sent on stream zero; replaced "AUTOMATIC_USE" on "CERTIFICATE" with "UNSOLICITED" on "USE_CERTIFICATE". (#482,#566)
Use Exported Requests from the TLS Exported Authenticators draft; eliminate facilities for expressing certificate requirements in "CERTIFICATE_REQUEST" frame. (#481)

A.5. Since draft-bishop-httpbis-http2-additional-certs-05:

o Adopted as draft-ietf-httpbis-http2-secondary-certs

Acknowledgements

Eric Rescorla pointed out several failings in an earlier revision. Andrei Popov contributed to the TLS considerations.

A substantial portion of Mike’s work on this draft was supported by Microsoft during his employment there.

Authors’ Addresses

Mike Bishop
Akamai

Email: mbishop@evequefou.be

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com
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.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (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/6265bis [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 October 29, 2019.
Internet-Draft  Cookies: HTTP State Management Mechanism      April 2019

5.1.3. Domain Matching ........................................... 19
5.1.4. Paths and Path-Match ..................................... 19
5.2. "Same-site" and "cross-site" Requests ...................... 20
  5.2.1. Document-based requests ................................. 21
  5.2.2. Worker-based requests .................................. 22
5.3. The Set-Cookie Header ....................................... 23
  5.3.1. The Expires Attribute .................................. 25
  5.3.2. The Max-Age Attribute .................................. 26
  5.3.3. The Domain Attribute .................................... 26
  5.3.4. The Path Attribute ...................................... 27
  5.3.5. The Secure Attribute .................................... 27
  5.3.6. The HttpOnly Attribute .................................. 27
  5.3.7. The SameSite Attribute .................................. 27
5.4. Storage Model ................................................ 28
5.5. The Cookie Header ............................................ 33
6. Implementation Considerations ................................. 35
  6.1. Limits ...................................................... 35
  6.2. Application Programming Interfaces ......................... 35
  6.3. IDNA Dependency and Migration ............................... 36
7. Privacy Considerations .......................................... 36
  7.1. Third-Party Cookies ......................................... 36
  7.2. User Controls ............................................... 37
  7.3. Expiration Dates ............................................ 37
8. Security Considerations ......................................... 38
  8.1. Overview .................................................... 38
  8.2. Ambient Authority .......................................... 38
  8.3. Clear Text .................................................. 39
  8.4. Session Identifiers ......................................... 40
  8.5. Weak Confidentiality ....................................... 40
  8.6. Weak Integrity .............................................. 41
  8.7. Reliance on DNS ............................................. 42
  8.8. SameSite Cookies ............................................ 42
     8.8.1. Defense in depth ..................................... 42
     8.8.2. Top-level Navigations ................................ 42
     8.8.3. Mashups and Widgets ................................... 43
     8.8.4. Server-controlled ..................................... 43
9. IANA Considerations ............................................. 43
  9.1. Cookie ...................................................... 43
  9.2. Set-Cookie .................................................. 44
10. References ..................................................... 44
  10.1. Normative References ....................................... 44
  10.2. Informative References ..................................... 45
  10.3. URIs ......................................................... 47
Appendix A. Changes ............................................... 48
  A.1. draft-ietf-httpbis-rfc6265bis-00 .......................... 48
  A.2. draft-ietf-httpbis-rfc6265bis-01 .......................... 48
  A.3. draft-ietf-httpbis-rfc6265bis-02 .......................... 49
  A.4. draft-ietf-httpbis-rfc6265bis-03 .......................... 49
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.

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 headers 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) rule is used where zero or more linear
whitespace characters MAY appear:

\[
\text{OWS} = *\{ [ \text{obs-fold} ] \text{WSP} \}
\]

\[
\text{obs-fold} = \text{CRLF}
\]

OWS SHOULD either not be produced or be produced as a single SP
character.
2.3. Terminology

The terms "user agent", "client", "server", "proxy", and "origin server" have the same meaning as in the HTTP/1.1 specification ([RFC7230], Section 2).

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 "request-target" as defined in Section 5.3 of [RFC7230].

Two sequences of octets are said to case-insensitively match each other if and only if they are equivalent under the i;j: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", "WorkerGlobalScope", "sandboxed origin browsing context flag", "parent browsing context", "shared worker", "the worker’s Documents", "nested browsing context", and "top-level browsing context" 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 4.2.1 of [RFC7231].

The term "public suffix" is defined in a note in Section 5.3 of [RFC6265] as "a domain that is controlled by a public registry", and are also known as "effective top-level domains" (eTLDs). For example, "example.com”’s public suffix is "com". User agents SHOULD use an up-to-date public suffix list, such as the one maintained by Mozilla at [PSL].

An origin’s "registered domain" is the origin’s host’s public suffix plus the label to its left. That is, for "https://www.example.com", the public suffix is "com", and the registered domain is "example.com". This concept is defined more rigorously in [PSL], and is also known as "effective top-level domain plus one" (eTLD+1).
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 in an HTTP response. In subsequent requests, the user agent returns a Cookie request header to the origin server. The Cookie header contains cookies the user agent received in previous Set-Cookie headers. The origin server is free to ignore the Cookie header or use its contents for an application-defined purpose.

Origin servers MAY send a Set-Cookie response header with any response. User agents MAY ignore Set-Cookie headers contained in responses with 100-level status codes but MUST process Set-Cookie headers contained in other responses (including responses with 400- and 500-level status codes). 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 3.2.2 of [RFC7230]) 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.

3.1. Examples

Using the Set-Cookie header, 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 example.com.

== Server -> User Agent ==
Set-Cookie: SID=31d4d96e407aad42; Path=/; Domain=example.com

== 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=example.com

== User Agent -> Server ==
Cookie: SID=31d4d96e407aad42; lang=en-US

Notice that the Cookie header 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 ==

== User Agent -> Server ==
Cookie: SID=31d4d96e407aad42; lang=en-US

Finally, to remove a cookie, the server returns a Set-Cookie header 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 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 headers.

4.1. Set-Cookie

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

4.1.1. Syntax

Informally, the Set-Cookie response header contains the header name "Set-Cookie" followed by a ":" and a cookie. Each cookie begins with a name-value-pair, followed by zero or more attribute-value pairs. Servers SHOULD NOT send Set-Cookie headers that fail to conform to the following grammar:
set-cookie-header = "Set-Cookie:" SP set-cookie-string
set-cookie-string = cookie-pair *( ";" SP cookie-av )
cookie-pair = cookie-name "=" cookie-value
token = <token, defined in [RFC7230], Section 3.2.6>
cookie-name = token
cookie-value = *cookie-octet / ( DQUOTE *cookie-octet DQUOTE )
cookie-octet = %x21 / %x23-2B / %x2D-3A / %x3C-5B / %x5D-7E
   ; US-ASCII characters excluding CTLs, 
   ; whitespace DQUOTE, comma, semicolon, 
   ; and backslash
token = <token, defined in [RFC7230], Section 3.2.6>
cookie-av = expires-av / max-age-av / domain-av / path-av / secure-av / httponly-av / samesite-av / extension-av
expires-av = "Expires=" sane-cookie-date
sane-cookie-date = <IMF-fixdate, defined in [RFC7231], Section 7.1.1.1>
max-age-av = "Max-Age=" non-zero-digit *DIGIT
   ; In practice, both expires-av and max-age-av
   ; are limited to dates representable by the
   ; user agent.
non-zero-digit = %x31-39
   ; digits 1 through 9
domain-av = "Domain=" domain-value
domain-value = <subdomain>
   ; defined in [RFC1034], Section 3.5, as
   ; enhanced by [RFC1123], Section 2.1
path-av = "Path=" path-value
path-value = *av-octet
secure-av = "Secure"
httponly-av = "HttpOnly"
samesite-av = "SameSite=" 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].
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 headers 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 5.4 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 5.3 for how user agents handle this case.)

If a server sends multiple responses containing Set-Cookie headers 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. 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, 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.

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.

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.

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 "example.com", the user agent will include the cookie in the Cookie header when making HTTP requests to example.com, www.example.com, and www.corp.example.com. (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 example.com returns a Set-Cookie header
without a Domain attribute, these user agents will erroneously send the cookie to www.example.com 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 "example.com" or of "foo.example.com" from foo.example.com, but the user agent will not accept a cookie with a Domain attribute of "bar.example.com" or of "baz.foo.example.com".

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 5.4 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]).

Although seemingly useful for protecting cookies from active network attackers, the Secure attribute protects only the cookie’s confidentiality. An active network attacker can overwrite Secure cookies from an insecure channel, disrupting their integrity (see Section 8.6 for more details).
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 (such as a web browser API that exposes cookies to scripts).

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://example.com/sekrit-image" will attach same-site cookies if and only if initiated from a context whose "site for cookies" is "example.com".

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 5.3.7.1. 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 treated as "None".

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 5.4.

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 would be rejected by a conformant user agent, as it does not have a "Secure" attribute.

Set-Cookie: __Secure-SID=12345; Domain=example.com

Whereas the following "Set-Cookie" header would be accepted:

Set-Cookie: __Secure-SID=12345; Domain=example.com; 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=example.com
Set-Cookie: __Host-SID=12345; Domain=example.com; Path=/
Set-Cookie: __Host-SID=12345; Secure; Domain=example.com; Path=/

While the would be accepted if set from a secure origin (e.g. "https://example.com/"), 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. 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 that conforms to the following grammar:

```plaintext
cookie-header = "Cookie:" OWS cookie-string OWS  
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.

Notice that the cookie attributes are not returned. In particular, the server cannot determine from the Cookie header 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 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, servers SHOULD NOT rely upon the serialization order. In particular, if the Cookie header 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.

### 5. User Agent Requirements

This section specifies the Cookie and Set-Cookie headers 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., for the sake of improved security); however, experiments have shown that such strictness reduces 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 headers.
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). For example, if the request-uri contains just a path (and optional query string), then the uri-path is that path (without the %x3F ("?") character or query string), and if the request-uri contains a full absoluteURI, the uri-path is the path component of that URI.

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

A request is "same-site" if its target’s URI’s origin’s registered domain is an exact match for the request’s client’s "site for cookies", or if the request has no client. The request is otherwise "cross-site".

For a given request ("request"), the following algorithm returns "same-site" or "cross-site":

1. If "request"’s client is "null", return "same-site".

   Note that this is the case for navigation triggered by the user directly (e.g. by typing directly into a user agent’s address bar).

2. Let "site" be "request"’s client’s "site for cookies" (as defined in the following sections).

3. Let "target" be the registered domain of "request"’s current url.

4. If "site" is an exact match for "target", return "same-site".

5. Return "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 registered domain of that URI’s origin represents the context in which a user most likely believes themselves to be interacting. We’ll label this domain the "top-level site".

For a document displayed in a top-level browsing context, we can stop here: the document’s "site for cookies" is the top-level site.

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 site if and only if the document and each of its ancestor documents’ origins have the same registered domain as the top-level site. Otherwise its "site for cookies" is the empty string.

Given a Document ("document"), the following algorithm returns its "site for cookies" (either a registered domain, or the empty string):

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"’s host’s registered domain is not an exact match for "top-origin"’s host’s registered domain, return the empty string.

5. Return "top-origin"’s host’s registered domain.
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 cookie" values, the worker’s "site for cookies" will be the empty string in cases where the values diverge, and the shared value in cases where the values agree.

Given a WorkerGlobalScope ("worker"), the following algorithm returns its "site for cookies" (either a registered domain, or the empty string):

1. Let "site" be "worker"’s origin’s host’s registered domain.
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 an exact match for "site", return the empty string.
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 registered domain of the Service Worker’s URI.

Given a ServiceWorkerGlobalScope ("worker"), the following algorithm returns its "site for cookies" (either a registered domain, or the empty string):

1. Return "worker"’s origin’s host’s registered domain.

5.3. The Set-Cookie Header

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. For example, the user agent might wish to block responses to "third-party" requests from setting cookies (see Section 7.1).

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. User agents use this algorithm so as to interoperate with servers that do not follow the recommendations in Section 4.

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 %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.

2. If the name-value-pair string lacks a `%x3D ("=")` character, ignore the set-cookie-string entirely.

3. The (possibly empty) 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 name string is empty, 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. Process the attribute-name and attribute-value according to the requirements in the following subsections. (Notice that attributes with unrecognized attribute-names are ignored.)

7. 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 5.4 for additional requirements triggered by receiving a cookie.)

5.3.1. 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. If the expiry-time is later than the last date the user agent can represent, the user agent MAY replace the expiry-time with the last representable date.

4. 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.

5. Append an attribute to the cookie-attribute-list with an attribute-name of Expires and an attribute-value of expiry-time.
5.3.2. 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. 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.
5. Append an attribute to the cookie-attribute-list with an attribute-name of Max-Age and an attribute-value of expiry-time.

5.3.3. The Domain Attribute

If the attribute-name case-insensitively matches the string "Domain", the user agent MUST process the cookie-av as follows.

1. If the attribute-value is empty, the behavior is undefined. However, the user agent SHOULD ignore the cookie-av entirely.
2. If the first character of the attribute-value string is %x2E ("."):
   1. Let cookie-domain be the attribute-value without the leading %x2E (".") character.
   Otherwise:
   1. Let cookie-domain be the entire attribute-value.
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.
5.3.4. 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.

5.3.5. 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.

5.3.6. 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.

5.3.7. 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 "None".

2. If cookie-av’s attribute-value is a case-insensitive match for "Strict", set "enforcement" to "Strict".

3. If cookie-av’s attribute-value is a case-insensitive match for "Lax", set "enforcement" to "Lax".

4. Append an attribute to the cookie-attribute-list with an attribute-name of "SameSite" and an attribute-value of "enforcement".
Note: This algorithm maps the "None" value, as well as any unknown value, to the "None" behavior, which is helpful for backwards compatibility when introducing new variants.

5.3.7.1. "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 [RFC7231] sense) HTTP method.

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 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. 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. For example, the user agent might wish to block receiving cookies
from "third-party" responses or the user agent might not wish to store cookies that exceed some size.

2. 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.

3. 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.

4. 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 an attribute-name of "Domain".

   Otherwise:
   1. Let the domain-attribute be the empty string.

5. 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. Ignore the cookie entirely and abort these steps.

NOTE: A "public suffix" is a domain that is controlled by a public registry, such as "com", "co.uk", and "pvt.k12.wy.us". This step is essential for preventing attacker.com from disrupting the integrity of example.com by setting a cookie with a Domain attribute of "com". Unfortunately, the set of public suffixes (also known as "registry controlled domains") changes over time. If feasible, user agents SHOULD use an up-to-date public suffix list, such as the one maintained by the Mozilla project at http://publicsuffix.org/ [4].

6. If the domain-attribute is non-empty:
   1. If the canonicalized request-host does not domain-match the domain-attribute:
      1. Ignore the cookie entirely and abort these steps.
      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.

7. 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 an attribute-name of "Path". Otherwise, set the cookie’s path to the default-path of the request-uri.

8. 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.
9. 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.

10. 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.

11. 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.

12. If the cookie’s secure-only-flag is not set, 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’.

13. If the cookie-attribute-list contains an attribute with an attribute-name of "SameSite", set the cookie’s same-site-flag to attribute-value (i.e. either "Strict", "Lax", or "None"). Otherwise, set the cookie’s same-site-flag to "None".

14. If the cookie’s "same-site-flag" is not "None", and the cookie is being set from a context whose "site for cookies" is not an exact match for request-uri’s host’s registered domain, then abort these steps and ignore the newly created cookie entirely.
15. 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.

16. 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 "/".

17. 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.

18. 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 not set, 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 date 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.

5.5. The Cookie Header

The user agent includes stored cookies in the Cookie HTTP request header.

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 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 send the cookie-string (defined below) as the value of the header field.

The user agent MUST use an algorithm equivalent to the following algorithm to compute the cookie-string from a cookie store and a request-uri:

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 request-host is identical to the cookie’s domain.
Or:

+ The cookie’s host-only-flag is false and the canonicalized request-host domain-matches the cookie’s domain.

* The request-uri’s path path-matches the cookie’s path.

* If the cookie’s secure-only-flag is true, then the request-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 cookie-string is being generated for a "non-HTTP" API (as defined by the user agent).

* If the cookie’s same-site-flag is not "None", and the HTTP request is cross-site (as defined in Section 5.2) then exclude the cookie unless all of the following statements hold:

1. The same-site-flag is "Lax"

2. The HTTP request’s method is "safe".

3. The HTTP request’s target browsing context 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. Output the cookie’s name, the %x3D (“=”) character, and the cookie’s value.
   2. 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:

   o At least 4096 bytes per cookie (as measured by the sum of the length of the cookie’s name, value, and attributes).
   o At least 50 cookies per domain.
   o At least 3000 cookies total.

   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 being included in every request.

   Servers SHOULD gracefully degrade if the user agent fails to return one or more cookies in the Cookie header because the user agent might evict any cookie at any time on orders from the user.

6.2. Application Programming Interfaces

   One reason the Cookie and Set-Cookie headers 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 headers, 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 are often criticized for letting servers track users. For example, a number of "web analytics" companies use cookies to recognize when a user returns to a web site or visits another web site. Although cookies are not the only mechanism servers can use to track users across HTTP requests, cookies facilitate tracking because they are persistent across user agent sessions and can be shared between hosts.

7.1. Third-Party Cookies

Particularly worrisome are so-called "third-party" cookies. In rendering an HTML document, a user agent often requests resources from other servers (such as advertising networks). These third-party servers can use cookies to track the user even if the user never visits the server directly. For example, if a user visits a site that contains content from a third party and then later visits another site that contains content from the same third party, the third party can track the user between the two sites.

Given this risk to user privacy, some user agents restrict how third-party cookies behave, and those restrictions vary widely. For instance, user agents might block third-party cookies entirely by
refusing to send Cookie headers or process Set-Cookie headers during third-party requests. They might take a less draconian approach by partitioning cookies based on the first-party context, sending one set of cookies to a given third party in one first-party context, and another to the same third party in another.

This document grants user agents wide latitude to experiment with third-party cookie policies that balance the privacy and compatibility needs of their users. However, this document does not endorse any particular third-party cookie policy.

Third-party cookie blocking policies are often ineffective at achieving their privacy goals if servers attempt to work around their restrictions to track users. In particular, two collaborating servers can often track users without using cookies at all by injecting identifying information into dynamic URLs.

### 7.2. 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 in outbound HTTP requests and the user agent MUST NOT process Set-Cookie headers in inbound HTTP responses.

Some user agents 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].

Some user agents provide users with the ability to approve individual writes to the cookie store. In many common usage scenarios, these controls generate a large number of prompts. However, some privacy-conscious users find these controls useful nonetheless.

### 7.3. 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 headers is transmitted in the clear.

1. All sensitive information conveyed in these headers is exposed to an eavesdropper.

2. A malicious intermediary could alter the headers as they travel in either direction, with unpredictable results.

3. A malicious client could alter the Cookie header 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 headers 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 the 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.example.com and bar.example.com. The foo.example.com server can set a cookie with a Domain attribute of "example.com" (possibly overwriting an existing "example.com" cookie set by bar.example.com), and the user agent will include that cookie in HTTP requests to bar.example.com. In the worst case, bar.example.com will be unable to distinguish this cookie from a cookie it set itself. The foo.example.com server might be able to leverage this ability to mount an attack against bar.example.com.

Even though the Set-Cookie header 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. For example, an HTTP response to a request for http://example.com/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 sent to https://example.com/ by impersonating a response from http://example.com/ and injecting a Set-Cookie header. The HTTPS server at example.com 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 example.com even if example.com uses HTTPS exclusively.

Servers can partially mitigate these attacks by encrypting and signing the contents of their cookies. However, using cryptography does not mitigate the issue completely because an attacker can replay a cookie he or she received from the authentic example.com 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://example.com/". They might expect that clicking on an emailed link to "https://projects.com/secret/project" would show them the secret project that they’re authorized to see, but if "projects.com" has marked their session cookies as "SameSite", then this cross-site navigation won’t send them along with the request. "projects.com" 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", and its absence would prompt a reauthentication step before executing any non-idempotent action. The former could drop the "SameSite" attribute entirely, or choose the "Lax" version of enforcement, in order to allow users access to data via top-level navigation.

8.8.3. Mashups and Widgets

The "SameSite" attribute is inappropriate for some important use-cases. In particular, note that content intended for embedding in a cross-site contexts (social networking widgets or commenting services, for instance) will not have access to same-site cookies. Cookies may be required for requests triggered in these cross-site contexts in order to provide seamless functionality that relies on a user’s state.

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.

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. Connection and/or socket pooling, Token Binding, and Channel ID all offer explicit methods of identification that servers could take advantage of.

9. IANA Considerations

The permanent message header field registry (see [RFC3864]) needs to be updated with the following registrations.

9.1. Cookie

Header field name: Cookie
Applicable protocol: http
Status: standard
Author/Change controller: IETF
Specification document: this specification (Section 5.5)
9.2. Set-Cookie

   Header field name: Set-Cookie

   Applicable protocol: http

   Status: standard

   Author/Change controller: IETF

   Specification document: this specification (Section 5.3)

10. References

10.1. Normative References


See Section 6.3 for an explanation why the normative reference to an obsoleted specification is needed.
10.2. Informative References

[Aggarwal2010]

[USASCII]


West, M., "Deprecate modification of 'secure' cookies from non-secure origins", draft-ietf-httpbis-cookie-alone-01 (work in progress), September 2016.

West, M., "Cookie Prefixes", draft-ietf-httpbis-cookie-prefixes-00 (work in progress), February 2016.


10.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/


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:

- 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 [7]

- Dropped "Cookie2" and "Set-Cookie2" from the IANA Considerations section: https://github.com/httpwg/http-extensions/issues/247 [8]

- 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:
  * Added the word "Cookies" to the document’s name:
  * Clarified that the "__Host-" prefix requires an explicit "Path"
    attribute: https://github.com/httpwg/http-extensions/issues/222
    [11]
  * Expanded the options for dealing with third-party cookies to
    include a brief mention of partitioning based on first-party:
  * Noted that double-quotes in cookie values are part of the
    value, and are not stripped: https://github.com/httpwg/http-
    extensions/issues/295 [13]
  * Fixed the "site for cookies" algorithm to return something that
    makes sense: https://github.com/httpwg/http-extensions/
    issues/302 [14]

A.4. draft-ietf-httpbis-rfc6265bis-03

- Clarified handling of invalid SameSite values:

- Reflect widespread implementation practice of including a cookie’s
  "host-only-flag" when calculating its uniqueness:
  https://github.com/httpwg/http-extensions/issues/199 [16]

- Introduced an explicit "None" value for the SameSite attribute:

Acknowledgements

This document is a minor update of RFC 6265, adding small features,
and aligning the specification with the reality of today’s
deployments. Here, we’re standing upon the shoulders of giants.
Authors’ Addresses

Adam Barth
Google, Inc

URI: https://www.adambarth.com/

Mike West
Google, Inc

Email: mkwst@google.com
URI: https://mikewest.org/
HTTP Representation Variants
draft-ietf-httpbis-variants-05

Abstract

This specification introduces an alternative way to communicate a secondary cache key for a HTTP resource, using the HTTP "Variants" and "Variant-Key" response header fields. Its aim is to make HTTP proactive content negotiation more cache-friendly.

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), 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/variants [3].

There is a prototype implementation of the algorithms herein at https://github.com/mnot/variants-toy [4].

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 September 26, 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 (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. Notational Conventions ............................. 4
2. The "Variants" HTTP Header Field .......................... 5
   2.1. Relationship to Vary ............................... 7
3. The "Variant-Key" HTTP Header Field ........................ 7
4. Cache Behaviour .......................................... 9
   4.1. Compute Possible Keys .............................. 10
   4.2. Check Vary ......................................... 11
   4.3. Example of Cache Behaviour .......................... 11
      4.3.1. A Variant Missing From the Cache .............. 12
      4.3.2. Variants That Don’t Overlap the Client’s Request 13
5. Origin Server Behaviour .................................. 13
   5.1. Examples ........................................... 14
      5.1.1. Single Variant ................................. 14
      5.1.2. Multiple Variants .............................. 15
      5.1.3. Partial Coverage ............................... 15
6. Defining Content Negotiation Using Variants ............... 16
7. IANA Considerations ...................................... 16
8. Security Considerations .................................. 17
9. References .............................................. 17
   9.1. Normative References ............................... 17
   9.2. Informative References ............................. 18
   9.3. URIs ............................................... 18
Appendix A. Variants for Existing Content Negotiation Mechanisms 19
   A.1. Accept ............................................. 19
   A.2. Accept-Encoding .................................... 20
   A.3. Accept-Language .................................... 20
Acknowledgements ........................................... 21
Author’s Address ........................................... 21
1. Introduction

HTTP proactive content negotiation ([RFC7231], Section 3.4.1) is seeing renewed interest, both for existing request headers like Accept-Language and for newer ones (for example, see [I-D.ietf-httpbis-client-hints]).

Successfully reusing negotiated responses that have been stored in a HTTP cache requires establishment of a secondary cache key ([RFC7234], Section 4.1). Currently, the Vary header ([RFC7231], Section 7.1.4) does this by nominating a set of request headers.

HTTP’s caching model allows a certain amount of latitude in normalising those request header field values, so as to increase the chances of a cache hit while still respecting the semantics of that header. However, normalisation is not formally defined, leading to divergence in cache behaviours.

Even when the headers’ semantics are understood, a cache does not know enough about the possible alternative representations available on the origin server to make an appropriate decision.

For example, if a cache has stored the following request/response pair:

```
GET /foo HTTP/1.1
Host: www.example.com
Accept-Language: en;q=0.5, fr;q=1.0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Language: en
Vary: Accept-Language
Transfer-Encoding: chunked

[English content]
```

Provided that the cache has full knowledge of the semantics of Accept-Language and Content-Language, it will know that an English representation is available and might be able to infer that a French representation is not available. But, it does not know (for example) whether a Japanese representation is available without making another request, incurring possibly unnecessary latency.

This specification introduces the HTTP Variants response header field (Section 2) to enumerate the available variant representations on the origin server, to provide clients and caches with enough information.
to properly satisfy requests - either by selecting a response from cache or by forwarding the request towards the origin - by following the algorithm defined in Section 4.

Its companion Variant-Key response header field (Section 3) indicates the applicable key(s) that the response is associated with, so that it can be reliably reused in the future. When this specification is in use, the example above might become:

GET /foo HTTP/1.1
Host: www.example.com
Accept-Language: en;q=0.5, fr;q=1.0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Language: en
Vary: Accept-Language
Variants: Accept-Language;de;en;jp
Variant-Key: en
Transfer-Encoding: chunked

[English content]

Proactive content negotiation mechanisms that wish to be used with Variants need to define how to do so explicitly; see Section 6. As a result, it is best suited for negotiation over request headers that are well-understood.

Variants also works best when content negotiation takes place over a constrained set of representations; since each variant needs to be listed in the header field, it is ill-suited for open-ended sets of representations.

Variants can be seen as a simpler version of the Alternates header field introduced by [RFC2295]; unlike that mechanism, Variants does not require specification of each combination of attributes, and does not assume that each combination has a unique URL.

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.
This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] but relies on Structured Headers from [I-D.ietf-httpbis-header-structure] for parsing.

Additionally, it uses the "field-name" rule from [RFC7230], and "type", "subtype", "content-coding" and "language-range" from [RFC7231].

2. The "Variants" HTTP Header Field

The Variants HTTP response header field indicates what representations are available for a given resource at the time that the response is produced, by enumerating the request header fields that it varies on, along with the values that are available for each.

Variants is a Structured Header [I-D.ietf-httpbis-header-structure]. Its value MUST be a list-of-lists (Section 3.3 of [I-D.ietf-httpbis-header-structure]) whose members are strings (Section 3.8 of [I-D.ietf-httpbis-header-structure]) or tokens (Section 3.9 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

```
Variants        = sh-list-of-lists
```

If Structured Header parsing fails or a list-member has the wrong type, the client MUST treat the representation as having no Variants header field.

The Variants header field represents an ordered list of "variant-axes", each of which consists of a request header "field-name" string and a list of "available-value" strings. Each inner-list in the Variants header field value is parsed into a variant-axis. The first list-member of the inner-list is interpreted as the field-name, and the remaining list-members are the available-values. Any list-member that is a token (Section 3.9 of [I-D.ietf-httpbis-header-structure]) is interpreted as a string containing the same characters.

Field-names in the Variants header field value MUST match the field-name production (Section 3.2 of [RFC7230]). Clients receiving an invalid field-name MUST NOT match it to any content negotiating mechanism.

So, given this example header field:

```
Variants: Accept-Encoding;gzip
```

a recipient can infer that the only content-coding available for that resource is "gzip" (along with the "identity" non-encoding; see Appendix A.2).
Given:

Variants: accept-encoding

A recipient can infer that no content-codings (beyond identity) are supported. Note that as always, field-name is case-insensitive.

A more complex example:

Variants: Accept-Encoding;gzip;br, Accept-Language;en ;fr

Here, recipients can infer that two content-codings in addition to "identity" are available, as well as two content languages. Note that, as with all Structured Header lists, they might occur in the same header field or separately, like this:

Variants: Accept-Encoding;gzip;bro
tli
Variants: Accept-Language;en ;fr

The ordering of available-values after the field-name is significant, as it might be used by the header’s algorithm for selecting a response (in this example, the first language is the default; see Appendix A.3).

The ordering of the request header fields themselves indicates descending application of preferences; in the example above, a cache that has all of the possible permutations stored will honour the client’s preferences for Accept-Encoding before honouring Accept-Language.

Origin servers SHOULD consistently send Variant header fields on all cacheable (as per [RFC7234], Section 3) responses for a resource, since its absence will trigger caches to fall back to Vary processing.

Likewise, servers MUST send the Variant-Key response header field when sending Variants, since its absence means that the stored response will not be reused when this specification is implemented.

_RFC EDITOR: Please remove the next paragraph before publication._

Implementations of drafts of this specification MUST implement an HTTP header field named "Variants-##" instead of the "Variants" header field specified by the final RFC, with "##" replaced by the draft number being implemented. For example, implementations of draft-ietf-httpbis-variants-05 would implement "Variants-05".
2.1. Relationship to Vary

This specification updates [RFC7234] to allow caches that implement it to ignore request header fields in the Vary header for the purposes of secondary cache key calculation ([RFC7234], Section 4.1) when their semantics are implemented as per this specification and their corresponding response header field is listed in Variants.

If any member of the Vary header does not have a corresponding variant that is understood by the implementation, it is still subject to the requirements there.

See Section 5.1.3 for an example.

In practice, implementation of Vary varies considerably. As a result, cache efficiency might drop considerably when Variants does not contain all of the headers referenced by Vary, because some implementations might choose to disable Variants processing when this is the case.

3. The "Variant-Key" HTTP Header Field

The Variant-Key HTTP response header field identifies a set of variants provided by the representation it occurs within. A variant is identified by a selection of one available-value from each variant-axis from the Variants header field.

Variant-Key is a Structured Header
[I-D.ietf-httpbis-header-structure]. Its value MUST be a list-of-lists (Section 3.3 of [I-D.ietf-httpbis-header-structure]) whose members are strings (Section 3.8 of [I-D.ietf-httpbis-header-structure]) or tokens (Section 3.9 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

Variant-Key = sh-list-of-lists

If Structured Header parsing fails or a list-member has the wrong type, the client MUST treat the representation as having no Variant-Key header field.

Each inner-list MUST have the same number of list-members as there are variant-axes in the representation’s Variants header field. If not, the client MUST treat the representation as having no Variant-Key header field.

Each list-member is treated as identifying an available-value for the corresponding variant-axis’ field-name. Any list-member that is a token (Section 3.9 of [I-D.ietf-httpbis-header-structure]) is
interpreted as a string containing the same characters. These available-values do not need to explicitly appear in the Variants header field. For example, Accept-Encoding defines an implicit "identity" available-value (Appendix A.2).

For example:

Variants: Accept-Encoding;gzip;br, Accept-Language;en ;fr
Variant-Key: gzip;fr

This header pair indicates that the representation has a "gzip" content-coding and "fr" content-language.

If the response can be used to satisfy more than one request, they can be listed in additional members. For example:

Variants: Accept-Encoding;gzip;br, Accept-Language;en ;fr
Variant-Key: gzip;fr, "identity";fr

indicates that this response can be used for requests whose Accept-Encoding algorithm selects "gzip" or "identity", as long as the Accept-Language algorithm selects "fr" - perhaps because there is no gzip-compressed French representation.

When more than one Variant-Key value is in a response, the first one present MUST correspond to the request that caused that response to be generated.

Parsing is strict. For example:

Variants: Accept-Encoding;gzip;br, Accept-Language;en ;fr
Variant-Key: gzip;fr, identity;fr, br;br;oops

is treated as if the Variant-Key header were completely absent, which will tend to disable caching for the representation that contains it.

Note that in

Variant-Key: gzip ;fr
Variant-Key: "gzip ";fr

The whitespace after "gzip" in the first header field value is excluded by the token parsing algorithm, but the whitespace in the second header field value is included by the string parsing algorithm. This will likely cause the second header field value to fail to match client requests.

_RFC EDITOR: Please remove the next paragraph before publication._
Implementations of drafts of this specification MUST implement an HTTP header field named "Variant-Key-##" instead of the "Variant-Key" header field specified by the final RFC, with "##" replaced by the draft number being implemented. For example, implementations of draft-ietf-httpbis-variants-05 would implement "Variant-Key-05".

4. Cache Behaviour

Caches that implement the Variants header field and the relevant semantics of the field-names it contains can use that knowledge to either select an appropriate stored representation, or forward the request if no appropriate representation is stored.

They do so by running this algorithm (or its functional equivalent) upon receiving a request:

Given incoming-request (a mapping of field-names to lists of field values), and stored-responses (a list of stored responses suitable for reuse as defined in Section 4 of [RFC7234], excepting the requirement to calculate a secondary cache key):

1. If stored-responses is empty, return an empty list.
2. Order stored-responses by the "Date" header field, most recent to least recent.
3. Let sorted-variants be an empty list.
4. If the freshest member of stored-responses (as per [RFC7234], Section 4.2) has one or more "Variants" header field(s) that successfully parse according to Section 2:
   1. Select one member of stored-responses with a "Variants" header field-value(s) that successfully parses according to Section 2 and let variants-header be this parsed value. This SHOULD be the most recent response, but MAY be from an older one as long as it is still fresh.
   2. For each variant-axis in variants-header:
      1. If variant-axis’ field-name corresponds to the request header field identified by a content negotiation mechanism that the implementation supports:
         1. Let request-value be the field-value associated with field-name in incoming-request (after being combined as allowed by Section 3.2.2 of [RFC7230]), or null if field-name is not in incoming-request.
2. Let sorted-values be the result of running the algorithm defined by the content negotiation mechanism with request-value and variant-axis' available-values.

3. Append sorted-values to sorted-variants.

   At this point, sorted-variants will be a list of lists, each member of the top-level list corresponding to a variant-axis in the Variants header field-value, containing zero or more items indicating available-values that are acceptable to the client, in order of preference, greatest to least.

5. Return result of running Compute Possible Keys (Section 4.1) on sorted-variants, an empty list and an empty list.

This returns a list of lists of strings suitable for comparing to the parsed Variant-Keys (Section 3) that represent possible responses on the server that can be used to satisfy the request, in preference order, provided that their secondary cache key (after removing the headers covered by Variants) matches. Section 4.2 illustrates one way to do this.

4.1. Compute Possible Keys

   This algorithm computes the cross-product of the elements of key-facets.

   Given key-facets (a list of lists of strings), and key-stub (a list of strings representing a partial key), and possible-keys (a list of lists of strings):

1. Let values be the first member of key-facets.

2. Let remaining-facets be a copy of all of the members of key-facets except the first.

3. For each value in values:

   1. Let this-key be a copy of key-stub.

   2. Append value to this-key.

   3. If remaining-facets is empty, append this-key to possible-keys.

   4. Otherwise, run Compute Possible Keys on remaining-facets, this-key and possible-keys.
4. Return possible-keys.

4.2. Check Vary

This algorithm is an example of how an implementation can meet the requirement to apply the members of the Vary header field that are not covered by Variants.

Given stored-response (a stored response):

1. Let filtered-vary be the field-value(s) of stored-response’s "Vary" header field.

2. Let processed-variants be a list containing the request header fields that identify the content negotiation mechanisms supported by the implementation.

3. Remove any member of filtered-vary that is a case-insensitive match for a member of processed-variants.

4. If the secondary cache key (as calculated in [RFC7234], Section 4.1) for stored_response matches incoming-request, using filtered-vary for the value of the "Vary" response header, return True.

5. Return False.

This returns a Boolean that indicates whether stored-response can be used to satisfy the request.

Note that implementation of the Vary header field varies in practice, and the algorithm above illustrates only one way to apply it. It is equally viable to forward the request if there is a request header listed in Vary but not Variants.

4.3. Example of Cache Behaviour

For example, if the selected variants-header was:

Variants: Accept-Language;en;fr;de, Accept-Encoding;gzip;br

and the request contained the headers:

Accept-Language: fr;q=1.0, en;q=0.1
Accept-Encoding: gzip

Then the sorted-variants would be:
Which means that the result of the Cache Behaviour algorithm would be:

```
[  
  ["fr", "gzip"],  
  ["fr", "identity"],  
  ["en", "gzip"],  
  ["en", "identity"]
]
```

Representing a first preference of a French, gzip’d response. Thus, if a cache has a response with:

```
Variant-Key: fr; gzip
```

it could be used to satisfy the first preference. If not, responses corresponding to the other keys could be returned, or the request could be forwarded towards the origin.

4.3.1. A Variant Missing From the Cache

If the selected variants-header was:

```
Variants: Accept-Language;en;fr;de
```

And a request comes in with the following headers:

```
Accept-Language: de;q=1.0, es;q=0.8
```

Then sorted-variants in Cache Behaviour is:

```
[  
  ["de"]  
]
```

If the cache contains responses with the following Variant-Keys:

```
Variant-Key: fr  
Variant-Key: en
```

Then the cache needs to forward the request to the origin server, since Variants indicates that "de" is available, and that is acceptable to the client.
4.3.2. Variants That Don’t Overlap the Client’s Request

If the selected variants-header was:

Variants: Accept-Language;en;fr;de

And a request comes in with the following headers:

Accept-Language: es;q=1.0, ja;q=0.8

Then sorted-variants in Cache Behaviour are:

```javascript
[ "en"
]
```

This allows the cache to return a "Variant-Key: en" response even though it’s not in the set the client prefers.

5. Origin Server Behaviour

Origin servers that wish to take advantage of Variants will need to generate both the Variants (Section 2) and Variant-Key (Section 3) header fields in all cacheable responses for a given resource. If either is omitted and the response is stored, it will have the effect of disabling caching for that resource until it is no longer stored (e.g., it expires, or is evicted).

Likewise, origin servers will need to assure that the members of both header field values are in the same order and have the same length, since discrepancies will cause caches to avoid using the responses they occur in.

The value of the Variants header should be relatively stable for a given resource over time; when it changes, it can have the effect of invalidating previously stored responses.

As per Section 2.1, the Vary header is required to be set appropriately when Variants is in use, so that caches that do not implement this specification still operate correctly.

Origin servers are advised to carefully consider which content negotiation mechanisms to enumerate in Variants; if a mechanism is not supported by a receiving cache, it will "downgrade" to Vary handling, which can negatively impact cache efficiency.
5.1. Examples

The operation of Variants is illustrated by the examples below.

5.1.1. Single Variant

Given a request/response pair:

GET /clancy HTTP/1.1
Host: www.example.com
Accept-Language: en;q=1.0, fr;q=0.5

HTTP/1.1 200 OK
Content-Type: image/gif
Content-Language: en
Cache-Control: max-age=3600
Variants: Accept-Language;en;de
Variant-Key: en
Vary: Accept-Language
Transfer-Encoding: chunked

Upon receipt of this response, the cache knows that two representations of this resource are available, one with a language of "en", and another whose language is "de".

Subsequent requests (while this response is fresh) will cause the cache to either reuse this response or forward the request, depending on what the selection algorithm determines.

So, if a request with "en" in Accept-Language is received and its q-value indicates that it is acceptable, the stored response is used. A request that indicates that "de" is acceptable will be forwarded to the origin, thereby populating the cache. A cache receiving a request that indicates both languages are acceptable will use the q-value to make a determination of what response to return.

A cache receiving a request that does not list either language as acceptable (or does not contain an Accept-Language at all) will return the "en" representation (possibly fetching it from the origin), since it is listed first in the Variants list.

Note that Accept-Language is listed in Vary, to assure backwards-compatibility with caches that do not support Variants.
5.1.2. Multiple Variants

A more complicated request/response pair:

GET /murray HTTP/1.1
Host: www.example.net
Accept-Language: en;q=1.0, fr;q=0.5
Accept-Encoding: gzip, br

HTTP/1.1 200 OK
Content-Type: image/gif
Content-Language: en
Content-Encoding: br
Variants: Accept-Language;en;jp;de
Variant-Key: en;br
Vary: Accept-Language, Accept-Encoding
Transfer-Encoding: chunked

Here, the cache knows that there are two axes that the response varies upon; language and encoding. Thus, there are a total of nine possible representations for the resource (including the identity encoding), and the cache needs to consider the selection algorithms for both axes.

Upon a subsequent request, if both selection algorithms return a stored representation, it can be served from cache; otherwise, the request will need to be forwarded to origin.

5.1.3. Partial Coverage

Now, consider the previous example, but where only one of the Vary’d axes (encoding) is listed in Variants:

GET /bar HTTP/1.1
Host: www.example.net
Accept-Language: en;q=1.0, fr;q=0.5
Accept-Encoding: gzip, br
HTTP/1.1 200 OK
Content-Type: image/gif
Content-Language: en
Content-Encoding: br
Variants: Accept-Encoding;br;gzip
Variant-Key: br
Vary: Accept-Language, Accept-Encoding
Transfer-Encoding: chunked

Here, the cache will need to calculate a secondary cache key as per [RFC7234], Section 4.1 - but considering only Accept-Language to be in its field-value - and then continue processing Variants for the set of stored responses that the algorithm described there selects.

6. Defining Content Negotiation Using Variants

To be usable with Variants, proactive content negotiation mechanisms need to be specified to take advantage of it. Specifically, they:

- MUST define a request header field that advertises the clients preferences or capabilities, whose field-name SHOULD begin with "Accept-".
- MUST define the syntax of an available-value that will occur in Variants and Variant-Key.
- MUST define an algorithm for selecting a result. It MUST return a list of available-values that are suitable for the request, in order of preference, given the value of the request header nominated above (or null if the request header is absent) and an available-values list from the Variants header. If the result is an empty list, it implies that the cache cannot satisfy the request.

Appendix A fulfils these requirements for some existing proactive content negotiation mechanisms in HTTP.

7. IANA Considerations

This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

- Header field name: Variants
- Applicable protocol: http
- Status: standard
This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

-.Header field name: Variant-Key
-Applicable protocol: http
-Status: standard

8. Security Considerations

If the number or advertised characteristics of the representations available for a resource are considered sensitive, the Variants header by its nature will leak them.

Note that the Variants header is not a commitment to make representations of a certain nature available; the runtime behaviour of the server always overrides hints like Variants.

9. References

9.1. Normative References

[I-D.ietf-httpbis-header-structure]


9.2. Informative References

[I-D.ietf-httpbis-client-hints]


9.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/


Appendix A. Variants for Existing Content Negotiation Mechanisms

This appendix defines the required information to use existing proactive content negotiation mechanisms (as defined in [RFC7231], Section 5.3) with the Variants header field.

A.1. Accept

This section defines variant handling for the Accept request header (section 5.3.2 of [RFC7231]).

The syntax of an available-value for Accept is:

```
accept-available-value = type "/" subtype
```

To perform content negotiation for Accept given a request-value and available-values:

1. Let preferred-available be an empty list.
2. Let preferred-types be a list of the types in the request-value (or the empty list if request-value is null), ordered by their weight, highest to lowest, as per Section 5.3.2 of [RFC7231] (omitting any coding with a weight of 0). If a type lacks an explicit weight, an implementation MAY assign one.
3. For each preferred-type in preferred-types:
   1. If any member of available-values matches preferred-type, using the media-range matching mechanism specified in Section 5.3.2 of [RFC7231] (which is case-insensitive), append those members of available-values to preferred-available (preserving the precedence order implied by the media ranges’ specificity).
4. If preferred-available is empty, append the first member of available-values to preferred-available. This makes the first available-value the default when none of the client’s preferences are available.
5. Return preferred-available.

Note that this algorithm explicitly ignores extension parameters on media types (e.g., "charset").
A.2. Accept-Encoding

This section defines variant handling for the Accept-Encoding request header (section 5.3.4 of [RFC7231]).

The syntax of an available-value for Accept-Encoding is:

accept-encoding-available-value = content-coding / "identity"

To perform content negotiation for Accept-Encoding given a request-value and available-values:

1. Let preferred-available be an empty list.
2. Let preferred-codings be a list of the codings in the request-value (or the empty list if request-value is null), ordered by their weight, highest to lowest, as per Section 5.3.1 of [RFC7231] (omitting any coding with a weight of 0). If a coding lacks an explicit weight, an implementation MAY assign one.
3. If "identity" is not a member of preferred-codings, append "identity".
4. Append "identity" to available-values.
5. For each preferred-coding in preferred-codings:
   1. If there is a case-insensitive, character-for-character match for preferred-coding in available-values, append that member of available-values to preferred-available.
6. Return preferred-available.

Note that the unencoded variant needs to have a Variant-Key header field with a value of "identity" (as defined in Section 5.3.4 of [RFC7231]).

A.3. Accept-Language

This section defines variant handling for the Accept-Language request header (section 5.3.5 of [RFC7231]).

The syntax of an available-value for Accept-Language is:

accept-encoding-available-value = language-range

To perform content negotiation for Accept-Language given a request-value and available-values:
1. Let preferred-available be an empty list.

2. Let preferred-langs be a list of the language-ranges in the request-value (or the empty list if request-value is null), ordered by their weight, highest to lowest, as per Section 5.3.1 of [RFC7231] (omitting any language-range with a weight of 0). If a language-range lacks a weight, an implementation MAY assign one.

3. For each preferred-lang in preferred-langs:
   1. If any member of available-values matches preferred-lang, using either the Basic or Extended Filtering scheme defined in Section 3.3 of [RFC4647], append those members of available-values to preferred-available (preserving their order).

4. If preferred-available is empty, append the first member of available-values to preferred-available. This makes the first available-value the default when none of the client’s preferences are available.

5. Return preferred-available.

Acknowledgements

This protocol is conceptually similar to, but simpler than, Transparent Content Negotiation [RFC2295]. Thanks to its authors for their inspiration.

It is also a generalisation of a Fastly VCL feature designed by Rogier ‘DocWilco’ Mulhuijzen.

Thanks to Hooman Beheshti, Ilya Grigorik, Leif Hedstrom, and Jeffrey Yasskin for their review and input.

Author’s Address

Mark Nottingham
Fastly

Email: mnot@mnot.net
URI: https://www.mnot.net/
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.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-http [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 October 25, 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 (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.  Notational Conventions .................................. 4
2. Connection Setup and Management ................................ 5
   2.1.  Draft Version Identification .............................. 5
   2.2.  Discovering an HTTP/3 Endpoint ........................................ 5
          2.2.1. QUIC Version Hints ....................................... 6
   2.3.  Connection Establishment ..................................... 6
   2.4.  Connection Reuse ............................................. 7
3. Stream Mapping and Usage ........................................ 7
   3.1.  Bidirectional Streams ....................................... 8
   3.2.  Unidirectional Streams ..................................... 8
          3.2.1. Control Streams ......................................... 10
          3.2.2. Push Streams ........................................... 10
          3.2.3. Reserved Stream Types .................................. 11
4. HTTP Framing Layer ............................................... 11
   4.1.  Frame Layout .............................................. 12
   4.2.  Frame Definitions .......................................... 13
          4.2.1. DATA ................................................. 13
          4.2.2. HEADERS .............................................. 14
          4.2.3. PRIORITY ............................................ 14
          4.2.4. CANCEL_PUSH ......................................... 17
          4.2.5. SETTINGS ........................................... 18
          4.2.6. PUSH_PROMISE ........................................ 20
          4.2.7. GOAWAY ............................................... 21
          4.2.8. MAX_PUSH_ID ........................................... 21
          4.2.9. DUPLICATE_PUSH ....................................... 22
          4.2.10. Reserved Frame Types ................................23
5. HTTP Request Lifecycle .......................................... 23
   5.1.  HTTP Message Exchanges ................................... 23
          5.1.1. Header Formatting and Compression .................... 25
          5.1.2. Request Cancellation and Rejection .................. 25
5.2. The CONNECT Method ........................................ 26
5.3. Prioritization ............................................. 27
  5.3.1. Placeholders ...................................... 28
  5.3.2. Priority Tree Maintenance ......................... 28
5.4. Server Push ............................................... 29
6. Connection Closure ......................................... 31
  6.1. Idle Connections ..................................... 31
  6.2. Connection Shutdown .................................. 31
  6.3. Immediate Application Closure ....................... 33
  6.4. Transport Closure .................................... 33
7. Extensions to HTTP/3 ........................................ 33
8. Error Handling ............................................. 34
  8.1. HTTP/3 Error Codes .................................. 34
9. Security Considerations .................................... 36
10. IANA Considerations ....................................... 37
  10.1. Registration of HTTP/3 Identification String ...... 37
  10.2. Registration of QUIC Version Hint Alt-Svc Parameter 37
  10.3. Frame Types .......................................... 37
  10.4. Settings Parameters ................................ 38
  10.5. Error Codes .......................................... 39
  10.6. Stream Types ........................................ 42
11. References .................................................. 43
  11.1. Normative References ................................ 43
  11.2. Informative References ............................... 44
  11.3. URIs .................................................. 45
Appendix A. Considerations for Transitioning from HTTP/2 ....... 45
  A.1. Streams ................................................ 45
  A.2. HTTP Frame Types .................................... 45
  A.3. HTTP/2 SETTINGS Parameters .......................... 48
  A.4. HTTP/2 Error Codes .................................. 49
Appendix B. Change Log ......................................... 50
  B.1. Since draft-ietf-quic-http-19 ....................... 50
  B.2. Since draft-ietf-quic-http-18 ....................... 50
  B.3. Since draft-ietf-quic-http-17 ....................... 50
  B.4. Since draft-ietf-quic-http-16 ....................... 51
  B.5. Since draft-ietf-quic-http-15 ....................... 51
  B.7. Since draft-ietf-quic-http-13 ....................... 52
  B.8. Since draft-ietf-quic-http-12 ....................... 52
  B.10. Since draft-ietf-quic-http-10 ...................... 52
  B.11. Since draft-ietf-quic-http-09 ...................... 52
  B.12. Since draft-ietf-quic-http-08 ...................... 53
  B.13. Since draft-ietf-quic-http-07 ...................... 53
  B.15. Since draft-ietf-quic-http-05 ...................... 53
  B.16. Since draft-ietf-quic-http-04 ...................... 53
  B.17. Since draft-ietf-quic-http-03 ...................... 54
1. Introduction

HTTP semantics are used for a broad range of services on the Internet. These semantics have commonly been used with two different TCP mappings, HTTP/1.1 and HTTP/2. HTTP/2 introduced a framing and multiplexing layer to improve latency without modifying the transport layer. However, TCP’s lack of visibility into parallel requests in both mappings limited the possible performance gains.

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, it has the capability to improve the performance of HTTP compared to a TCP mapping. QUIC also incorporates TLS 1.3 at the transport layer, offering comparable security to running TLS over TCP, but with improved connection setup latency (unless TCP Fast Open [RFC7413] is used).

This document defines a mapping of HTTP semantics over the QUIC transport protocol, drawing heavily on the design of HTTP/2. This document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [RFC7540].

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] when, and only when, they appear in all capitals, as shown here.

Field definitions are given in Augmented Backus-Naur Form (ABNF), as defined in [RFC5234].

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].
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 4.2.

2. Connection Setup and Management

2.1. Draft Version Identification

*RFC Editor's Note:* Please remove this section prior to publication of a final version of this document.

HTTP/3 uses the token "h3" to identify itself in ALPN and Alt-Svc. Only implementations of the final, published RFC can identify themselves as "h3". Until such an RFC exists, implementations MUST NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "h3-01".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 which reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "h3-09-rickroll". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

2.2. Discovering an HTTP/3 Endpoint

An HTTP origin advertises 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 ALPN token defined in Section 2.3.

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 and, if successful, send HTTP requests using the mapping described in this document.
Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

Servers MAY serve HTTP/3 on any UDP port, since an alternative always includes an explicit port.

2.2.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/3 clients. QUIC versions are four-byte sequences with no additional constraints on format. Leading zeros SHOULD be omitted for brevity.

Syntax:

quic = DQUOTE version-number ["," version-number] * DQUOTE
version-number = 1*8HEXDIG; hex-encoded QUIC version

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Reserved versions MAY be listed, but unreserved versions which are not supported by the alternative SHOULD NOT be present in the list. Origins MAY omit supported versions for any reason.

Clients MUST ignore any included versions which they do not support. The "quic" parameter MUST NOT occur more than once; clients SHOULD process only the first occurrence.

For example, suppose a server supported both version 0x00000001 and the version rendered in ASCII as "Q034". If it also opted to include the reserved version (from Section 15 of [QUIC-TRANSPORT]) 0x1abadaba, it could specify the following header field:

Alt-Svc: h3=":49288";quic="1,1abadaba,51303334"

A client acting on this header field would drop the reserved version (not supported), then attempt to connect to the alternative using the first version in the list which it does support, if any.

2.3. Connection Establishment

HTTP/3 relies on QUIC as the underlying transport. The QUIC version being used MUST use TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients MUST indicate the target domain name during
the TLS handshake. This may be done using the Server Name Indication (SNI) [RFC6066] extension to TLS or using some other mechanism.

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 4.2.5) MUST be sent by each endpoint as the initial frame of their respective HTTP control stream (see Section 3.2.1).

2.4. Connection Reuse

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. The client MAY send any requests for which the client considers the server authoritative.

An authoritative HTTP/3 endpoint is typically discovered because the client has received an Alt-Svc record from the request’s origin which nominates the endpoint as a valid HTTP Alternative Service for that origin. As required by [RFC7838], clients MUST check that the nominated server can present a valid certificate for the origin before considering it authoritative. Clients MUST NOT assume that an HTTP/3 endpoint is authoritative for other origins without an explicit signal.

A server that does not wish clients to reuse 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 9.1.2 of [RFC7540]).

The considerations discussed in Section 9.1 of [RFC7540] also apply to the management of HTTP/3 connections.

3. 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. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received QUIC STREAM frames, exposing the data contained within as 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 headers 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 connection context.

3.1. 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. 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 client SHOULD send non-zero values for the QUIC transport parameters "initial_max_stream_data_bidi_local". An HTTP/3 server SHOULD send non-zero values for the QUIC transport parameters "initial_max_stream_data_bidi_remote" and "initial_max_bidi_streams". It is recommended that "initial_max_bidi_streams" be no smaller than 100, so as to not unnecessarily limit parallelism.

These streams carry frames related to the request/response (see Section 5.1). When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error (see HTTP_MALFORMED_FRAME in Section 8.1). Streams which terminate abruptly may be reset at any point in the frame.

HTTP/3 does not use server-initiated bidirectional streams; clients MUST omit or specify a value of zero for the QUIC transport parameter "initial_max_bidi_streams".

3.2. 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.
Some stream types are reserved (Section 3.2.3). Two stream types are defined in this document: control streams (Section 3.2.1) and push streams (Section 3.2.2). Other stream types can be defined by extensions to HTTP/3; see Section 7 for more details.

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 set low values for the QUIC transport parameters "initial_max_uni_streams" and "initial_max_stream_data_uni" will increase the chance that the remote peer reaches the limit early and becomes blocked. In particular, the value chosen for "initial_max_uni_streams" should consider that remote peers may wish to exercise reserved stream behaviour (Section 3.2.3). To reduce the likelihood of blocking, both clients and servers SHOULD send a value of three or greater for the QUIC transport parameter "initial_max_uni_streams", and a value of 1,024 or greater for the QUIC transport parameter "initial_max_stream_data_uni".

If the stream header indicates a stream type which 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 trigger a QUIC STOP_SENDING frame with an error code of HTTP_UNKNOWN_STREAM_TYPE, but MUST NOT consider such streams to be an error of any kind.

Implementations MAY send stream types before knowing whether the peer supports them. However, stream types which 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.
3.2.1. 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 4.2.

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 HTTP_MISSING_SETTINGS. Only one control stream per peer is permitted; receipt of a second stream which claims to be a control stream MUST be treated as a connection error of type HTTP_WRONG_STREAM_COUNT. The sender MUST NOT close the control stream. If the control stream is closed at any point, this MUST be treated as a connection error of type HTTP_CLOSED_CRITICAL_STREAM.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon they are able. Depending on whether 0-RTT is enabled on the connection, either client or server might be able to send stream data first after the cryptographic handshake completes.

3.2.2. Push Streams

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 4.2, and fulfills a promised server push. Server push and Push IDs are described in Section 5.4.

Only servers can push; if a server receives a client-initiated push stream, this MUST be treated as a stream error of type HTTP_WRONG_STREAM_DIRECTION.

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 HTTP_DUPLICATE_PUSH.
3.2.3. Reserved Stream Types

Stream types of the format "0x1f * N + 0x21" for 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 implementation chooses.

4. HTTP Framing Layer

HTTP frames are carried on QUIC streams, as described in Section 3. HTTP/3 defines three stream types: control stream, request stream, and push stream. This section describes HTTP/3 frame formats and the streams types on which they are permitted; see Table 1 for an overview. A comparison between HTTP/2 and HTTP/3 frames is provided in Appendix A.2.
<table>
<thead>
<tr>
<th>Frame</th>
<th>Control Stream</th>
<th>Request Stream</th>
<th>Push Stream</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Section 4.2.1</td>
</tr>
<tr>
<td>HEADERS</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Section 4.2.2</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>Yes</td>
<td>Yes (1)</td>
<td>No</td>
<td>Section 4.2.3</td>
</tr>
<tr>
<td>CANCEL_PUSH</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Section 4.2.4</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>Yes (1)</td>
<td>No</td>
<td>No</td>
<td>Section 4.2.5</td>
</tr>
<tr>
<td>PUSH_PROMISE</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Section 4.2.6</td>
</tr>
<tr>
<td>GOAWAY</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Section 4.2.7</td>
</tr>
<tr>
<td>MAX_PUSH_ID</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Section 4.2.8</td>
</tr>
<tr>
<td>DUPLICATE_PUSH</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Section 4.2.9</td>
</tr>
</tbody>
</table>

Table 1: HTTP/3 frames and stream type overview

Certain frames can only occur as the first frame of a particular stream type; these are indicated in Table 1 with a (1). Specific guidance is provided in the relevant section.

4.1. Frame Layout

All frames have the following 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 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 HTTP_MALFORMED_FRAME.

4.2. Frame Definitions

4.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of bytes associated with an HTTP request or response payload.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on either control stream, the recipient MUST respond with a connection error (Section 8) of type HTTP_WRONG_STREAM.
4.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry a header block, compressed using QPACK. See [QPACK] for more details.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Header Block (*)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 5: HEADERS frame payload

HEADERS frames can only be sent on request / push streams.

4.2.3. PRIORITY

The PRIORITY (type=0x2) frame specifies the client-advised priority of a request, server push or placeholder.

A PRIORITY frame identifies an element to prioritize, and an element upon which it depends. A Prioritized ID or Dependency ID identifies a client-initiated request using the corresponding stream ID, a server push using a Push ID (see Section 4.2.6), or a placeholder using a Placeholder ID (see Section 5.3.1).

When a client initiates a request, a PRIORITY frame MAY be sent as the first frame of the stream, creating a dependency on an existing element. In order to ensure that prioritization is processed in a consistent order, any subsequent PRIORITY frames for that request MUST be sent on the control stream. A PRIORITY frame received after other frames on a request stream MUST be treated as a stream error of type HTTP_UNEXPECTED_FRAME.

If, by the time a new request stream is opened, its priority information has already been received via the control stream, the PRIORITY frame sent on the request stream MUST be ignored.
The PRIORITY frame payload has the following fields:

PT (Prioritized Element Type): A two-bit field indicating the type of element being prioritized (see Table 2). When sent on a request stream, this MUST be set to "11". When sent on the control stream, this MUST NOT be set to "11".

DT (Element Dependency Type): A two-bit field indicating the type of element being depended on (see Table 3).

Empty: A four-bit field which MUST be zero when sent and has no semantic value on receipt.

Prioritized Element ID: A variable-length integer that identifies the element being prioritized. Depending on the value of Prioritized Type, this contains the Stream ID of a request stream, the Push ID of a promised resource, a Placeholder ID of a placeholder, or is absent.

Element Dependency ID: A variable-length integer that identifies the element on which a dependency is being expressed. Depending on the value of Dependency Type, this contains the Stream ID of a request stream, the Push ID of a promised resource, the Placeholder ID of a placeholder, or is absent. For details of dependencies, see Section 5.3 and [RFC7540], Section 5.3.

Weight: An unsigned 8-bit integer representing a priority weight for the prioritized element (see [RFC7540], Section 5.3). Add one to the value to obtain a weight between 1 and 256.

The values for the Prioritized Element Type (Table 2) and Element Dependency Type (Table 3) imply the interpretation of the associated Element ID fields.
Table 2: Prioritized Element Types

<table>
<thead>
<tr>
<th>PT Bits</th>
<th>Type Description</th>
<th>Prioritized Element ID Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Request stream</td>
<td>Stream ID</td>
</tr>
<tr>
<td>01</td>
<td>Push stream</td>
<td>Push ID</td>
</tr>
<tr>
<td>10</td>
<td>Placeholder</td>
<td>Placeholder ID</td>
</tr>
<tr>
<td>11</td>
<td>Current stream</td>
<td>Absent</td>
</tr>
</tbody>
</table>

Table 3: Element Dependency Types

<table>
<thead>
<tr>
<th>DT Bits</th>
<th>Type Description</th>
<th>Element Dependency ID Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Request stream</td>
<td>Stream ID</td>
</tr>
<tr>
<td>01</td>
<td>Push stream</td>
<td>Push ID</td>
</tr>
<tr>
<td>10</td>
<td>Placeholder</td>
<td>Placeholder ID</td>
</tr>
<tr>
<td>11</td>
<td>Root of the tree</td>
<td>Absent</td>
</tr>
</tbody>
</table>

Note that unlike in [RFC7540], the root of the tree cannot be referenced using a Stream ID of 0, as in QUIC stream 0 carries a valid HTTP request. The root of the tree cannot be reprioritized. A PRIORITY frame sent on a request stream with the Prioritized Element Type set to any value other than "11" or which expresses a dependency on a request with a greater Stream ID than the current stream MUST be treated as a stream error of type HTTP_MALFORMED_FRAME. Likewise, a PRIORITY frame sent on a control stream with the Prioritized Element Type set to "11" MUST be treated as a connection error of type HTTP_MALFORMED_FRAME. A PRIORITY frame with Empty bits not set to zero MAY be treated as a connection error of type HTTP_MALFORMED_FRAME.

When a PRIORITY frame claims to reference a request, the associated ID MUST identify a client-initiated bidirectional stream. A server MUST treat receipt of a PRIORITY frame identifying a stream of any other type as a connection error of type HTTP_MALFORMED_FRAME.

A PRIORITY frame that references a non-existent Push ID, a Placeholder ID greater than the server’s limit, or a Stream ID the
client is not yet permitted to open MUST be treated as an HTTP_LIMIT_EXCEEDED error.

A PRIORITY frame received on any stream other than a request or control stream MUST be treated as a connection error of type HTTP_WRONG_STREAM.

PRIORITY frames received by a client MUST be treated as a connection error of type HTTP_UNEXPECTED_FRAME.

4.2.4.  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 4.2.6), encoded as a variable-length integer.

When a server receives this frame, it aborts sending the response for the identified server push. If the server has not yet started to send the server push, it can use the receipt of a CANCEL_PUSH frame to avoid opening a push stream. If the push stream has been opened by the server, the server SHOULD send a QUIC RESET_STREAM frame on that stream and cease transmission of the response.

A server can send the CANCEL_PUSH frame to indicate that it will not be fulfilling a promise prior to creation of a push stream. Once the push stream has been created, sending CANCEL_PUSH has no effect on the state of the push stream. A QUIC RESET_STREAM frame SHOULD be used instead to abort transmission of the server push response.

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 stream error of type HTTP_WRONG_STREAM.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Figure 7: CANCEL_PUSH frame payload

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 4.2.6).

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.
4.2.5. 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 a connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of each control stream (see Section 3.2.1) by each peer, and MUST NOT be sent subsequently or on any other stream. If an endpoint receives a SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type HTTP_WRONG_STREAM. If an endpoint receives a second SETTINGS frame, the endpoint MUST respond with a connection error of type HTTP_UNEXPECTED_FRAME.

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which 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 header, while servers are more cautious about request size.

Parameters MUST NOT occur more than once in the SETTINGS frame. A receiver MAY treat the presence of the same parameter more than once as a connection error of type HTTP_MALFORMED_FRAME.

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.

```
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Identifier (i)                     ...                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Value (i)                          ...                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: SETTINGS parameter format
An implementation MUST ignore the contents for any SETTINGS identifier it does not understand.

4.2.5.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/3:

- **SETTINGS_MAX_HEADER_LIST_SIZE (0x6):** The default value is unlimited. See Section 5.1.1 for usage.

- **SETTINGS_NUM_PLACEHOLDERS (0x9):** The default value is 0. However, this value SHOULD be set to a non-zero value by servers. See Section 5.3.1 for usage.

Setting identifiers of the format "0x1f * N + 0x21" for 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.

Additional settings can be defined by extensions to HTTP/3; see Section 7 for more details.

4.2.5.2. Initialization

An HTTP implementation MUST NOT send frames or requests which would be invalid based on its current understanding of the peer’s settings. All settings begin at an initial value, and are updated upon receipt of a SETTINGS frame. 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. 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 MUST store the settings the server provided in the session being resumed and MUST comply with stored settings until the current server settings are received.

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.
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.

4.2.6. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x5) is used to carry a promised request header set from server to client on a request stream, as in HTTP/2.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Header Block (*)                      ...|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 9: PUSH_PROMISE frame payload

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 5.4), CANCEL_PUSH frames (Section 4.2.4), DUPLICATE_PUSH frames (Section 4.2.9), and PRIORITY frames (Section 4.2.3).

Header Block: QPACK-compressed 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 4.2.8) and MUST NOT use the same Push ID in multiple PUSH_PROMISE frames. A client MUST treat receipt of a PUSH_PROMISE that contains a larger Push ID than the client has advertised or a Push ID which has already been promised as a connection error of type HTTP_MALFORMED_FRAME.

If a PUSH_PROMISE frame is received on either control stream, the recipient MUST respond with a connection error (Section 8) of type HTTP_WRONG_STREAM.

See Section 5.4 for a description of the overall server push mechanism.
4.2.7. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of a connection by a server. GOAWAY allows a server to stop accepting new requests while still finishing processing of previously received requests. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

Figure 10: GOAWAY frame payload

The GOAWAY frame is always sent on the control stream. 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 HTTP_WRONG_STREAM.

Clients do not need to send GOAWAY to initiate a graceful shutdown; they simply stop making new requests. A server MUST treat receipt of a GOAWAY frame on any stream as a connection error (Section 8) of type HTTP_UNEXPECTED_FRAME.

The GOAWAY frame applies to the connection, not a specific stream. A client MUST treat a GOAWAY frame on a stream other than the control stream as a connection error (Section 8) of type HTTP_UNEXPECTED_FRAME.

See Section 6.2 for more information on the use of the GOAWAY frame.

4.2.8. 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 a PUSH_PROMISE frame. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit set by the QUIC MAX_STREAMS frame.

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 HTTP_WRONG_STREAM.
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 HTTP_UNEXPECTED_FRAME.

The maximum Push ID is unset when a 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.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 11: MAX_PUSH_ID frame payload

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 4.2.6). A MAX_PUSH_ID frame cannot reduce the maximum Push ID; receipt of a MAX_PUSH_ID that contains a smaller value than previously received MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

4.2.9. DUPLICATE_PUSH

The DUPLICATE_PUSH frame (type=0xE) is used by servers to indicate that an existing pushed resource is related to multiple client requests.

The DUPLICATE_PUSH frame is always sent on a request stream. Receipt of a DUPLICATE_PUSH frame on any other stream MUST be treated as a connection error of type HTTP_WRONG_STREAM.

A client MUST NOT send a DUPLICATE_PUSH frame. A server MUST treat the receipt of a DUPLICATE_PUSH frame as a connection error of type HTTP_UNEXPECTED_FRAME.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12: DUPLICATE_PUSH frame payload
The DUPLICATE_PUSH frame carries a single variable-length integer that identifies the Push ID of a resource that the server has previously promised (see Section 4.2.6).

This frame allows the server to use the same server push in response to multiple concurrent requests. Referencing the same server push ensures that a promise can be made in relation to every response in which server push might be needed without duplicating request headers or pushed responses.

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 DUPLICATE_PUSH that uses a Push ID that they have since consumed and discarded are forced to ignore the DUPLICATE_PUSH.

4.2.10. Reserved Frame Types

Frame types of the format "0x1f * N + 0x21" for integer values of N are reserved to exercise the requirement that unknown types be ignored (Section 7). These frames 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 frames to have any meaning upon receipt.

The payload and length of the frames are selected in any manner the implementation chooses.

5. HTTP Request Lifecycle

5.1. HTTP Message Exchanges

A client sends an HTTP request on a client-initiated bidirectional QUIC stream. A client MUST send only a single request on a given stream. A server sends one or more HTTP responses on the same stream as the request, as detailed below.

An HTTP message (request or response) consists of:

1. the message header (see [RFC7230], Section 3.2), sent as a single HEADERS frame (see Section 4.2.2),

2. the payload body (see [RFC7230], Section 3.3), sent as a series of DATA frames (see Section 4.2.1),
3. optionally, one HEADERS frame containing the trailer-part, if
   present (see [RFC7230], Section 4.1.2).

A server MAY interleave one or more PUSH_PROMISE frames (see
   Section 4.2.6) with the frames of a response message. These
   PUSH_PROMISE frames are not part of the response; see Section 5.4 for
   more details.

The "chunked" transfer encoding defined in Section 4.1 of [RFC7230]
   MUST NOT be used.

Trailing header fields are carried in an additional HEADERS frame
   following the body. Senders MUST send only one HEADERS frame in the
   trailers section; receivers MUST discard any subsequent HEADERS
   frames.

A response MAY consist of multiple messages when and only when one or
   more informational responses (1xx, see [RFC7231], Section 6.2)
   precede a final response to the same request. Non-final responses do
   not contain a payload body or trailers.

An HTTP request/response exchange fully consumes a bidirectional QUIC
   stream. After sending a request, a client MUST close the stream for
   sending. Unless using the CONNECT method (see Section 5.2), 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 an 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 stream terminates without
   enough of the HTTP message to provide a complete response, the server
   SHOULD abort its response with the error code
   HTTP_INCOMPLETE_REQUEST.

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 this is true, a
   server MAY request that the client abort transmission of a request
   without error by triggering a QUIC STOP_SENDING frame with error code
   HTTP_EARLY_RESPONSE, sending a complete response, and cleanly closing
   its 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.
5.1.1. Header Formatting and Compression

HTTP message headers carry information as a series of key-value pairs, called header fields. For a listing of registered HTTP header fields, see the "Message Header Field" registry maintained at https://www.iana.org/assignments/message-headers [4].

Just as in previous versions of HTTP, header field names are strings of ASCII characters that are compared in a case-insensitive fashion. Properties of HTTP header field names and values are discussed in more detail in Section 3.2 of [RFC7230], though the wire rendering in HTTP/3 differs. As in HTTP/2, header field names MUST be converted to lowercase prior to their encoding. A request or response containing uppercase header field names MUST be treated as malformed.

As in HTTP/2, HTTP/3 uses special pseudo-header fields beginning with the ‘:’ character (ASCII 0x3a) to convey the target URI, the method of the request, and the status code for the response. These pseudo-header fields are defined in Section 8.1.2.3 and 8.1.2.4 of [RFC7540]. Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in [RFC7540]. The restrictions on the use of pseudo-header fields in Section 8.1.2.1 of [RFC7540] also apply to HTTP/3.

HTTP/3 uses QPACK header compression as described in [QPACK], a variation of HPACK which allows the flexibility to avoid header-compression-induced head-of-line blocking. See that document for additional details.

An HTTP/3 implementation MAY impose a limit on the maximum size of the header it will accept on an individual HTTP message; encountering a larger message header SHOULD be treated as a stream error of type "HTTP_EXCESSIVE_LOAD". If an implementation wishes to advise its peer of this limit, it can be conveyed as a number of bytes in the "SETTINGS_MAX_HEADER_LIST_SIZE" parameter. The size of a header list is calculated based on the uncompressed size of header fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each header field.

5.1.2. Request Cancellation and Rejection

Clients can cancel requests by aborting the stream (QUIC RESET_STREAM and/or STOP_SENDING frames, as appropriate) with an error code of HTTP_REQUEST_CANCELLED (Section 8.1). When the client cancels a response, it indicates that this response is no longer of interest. Implementations SHOULD cancel requests by aborting both directions of a stream.
When the server rejects a request without performing any application processing, it SHOULD abort its response stream with the error code HTTP_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 on a new connection. Servers MUST NOT use the HTTP_REQUEST_REJECTED error code for requests which were partially or fully processed. When a server abandons a response after partial processing, it SHOULD abort its response stream with the error code HTTP_REQUEST_CANCELLED.

When a client sends a STOP_SENDING with HTTP_REQUEST_CANCELLED, a server MAY send the error code HTTP_REQUEST_REJECTED in the corresponding RESET_STREAM if no processing was performed. Clients MUST NOT reset streams with the HTTP_REQUEST_REJECTED error code except in response to a QUIC STOP_SENDING frame that contains the same code.

If a stream is cancelled 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. Automatically retrying such requests is not possible, unless this is otherwise permitted (e.g., idempotent actions like GET, PUT, or DELETE).

5.2. The CONNECT Method

The pseudo-method CONNECT ([RFC7231], Section 4.3.6) 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, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes.

A CONNECT request in HTTP/3 functions in the same manner as in HTTP/2. The request MUST be formatted as described in [RFC7540], Section 8.3. A CONNECT request that does not conform to these restrictions is malformed. The request stream MUST NOT be closed at the end of the request.

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 [RFC7231], Section 4.3.6.
All DATA frames on the stream correspond to data sent or received on the TCP connection. 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.

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 terminate the send stream that it sends to the client. TCP connections which 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 with QUIC RESET_STREAM frame. 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 HTTP_CONNECT_ERROR (Section 8.1). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the QUIC connection.

5.3. Prioritization

HTTP/3 uses a priority scheme similar to that described in [RFC7540], Section 5.3. In this priority scheme, a given element can be designated as dependent upon another element. This information is expressed in the PRIORITY frame Section 4.2.3 which identifies the element and the dependency. The elements that can be prioritized are:

- Requests, identified by the ID of the request stream
- Pushes, identified by the Push ID of the promised resource (Section 4.2.6)
- Placeholders, identified by a Placeholder ID

Taken together, the dependencies across all prioritized elements in a connection form a dependency tree. An element can depend on another element or on the root of the tree. A reference to an element which is no longer in the tree is treated as a reference to the root of the tree. The structure of the dependency tree changes as PRIORITY frames modify the dependency links between prioritized elements.
Due to reordering between streams, an element can also be prioritized which is not yet in the tree. Such elements are added to the tree with the requested priority.

When a prioritized element is first created, it has a default initial weight of 16 and a default dependency. Requests and placeholders are dependent on the root of the priority tree; pushes are dependent on the client request on which the PUSH_PROMISE frame was sent.

Requests may override the default initial values by including a PRIORITIY frame (see Section 4.2.3) at the beginning of the stream. These priorities can be updated by sending a PRIORITY frame on the control stream.

5.3.1. Placeholders

In HTTP/2, certain implementations used closed or unused streams as placeholders in describing the relative priority of requests. This created confusion as servers could not reliably identify which elements of the priority tree could be discarded safely. Clients could potentially reference closed streams long after the server had discarded state, leading to disparate views of the prioritization the client had attempted to express.

In HTTP/3, a number of placeholders are explicitly permitted by the server using the "SETTINGS_NUM_PLACEHOLDERS" setting. Because the server commits to maintaining these placeholders in the prioritization tree, clients can use them with confidence that the server will not have discarded the state. Clients MUST NOT send the "SETTINGS_NUM_PLACEHOLDERS" setting; receipt of this setting by a server MUST be treated as a connection error of type "HTTP_WRONG_SETTING_DIRECTION".

Placeholders are identified by an ID between zero and one less than the number of placeholders the server has permitted.

Like streams, placeholders have priority information associated with them.

5.3.2. Priority Tree Maintenance

Because placeholders will be used to "root" any persistent structure of the tree which the client cares about retaining, servers can aggressively prune inactive regions from the priority tree. For prioritization purposes, a node in the tree is considered "inactive" when the corresponding stream has been closed for at least two round-trip times (using any reasonable estimate available on the server). This delay helps mitigate race conditions where the server has pruned
a node the client believed was still active and used as a Stream Dependency.

Specifically, the server MAY at any time:

- Identify and discard branches of the tree containing only inactive nodes (i.e. a node with only other inactive nodes as descendants, along with those descendants)

- Identify and condense interior regions of the tree containing only inactive nodes, allocating weight appropriately

```
    x    x    x
     |     |     |
    P     P     P
   / \   / \   / \
   I I ==>  I  ==>  A
  / \   / \   / \
 A   I   A   A
   |   |   |   |
   A   A   A
```

**Figure 13: Example of Priority Tree Pruning**

In the example in Figure 13, "P" represents a Placeholder, "A" represents an active node, and "I" represents an inactive node. In the first step, the server discards two inactive branches (each a single node). In the second step, the server condenses an interior inactive node. Note that these transformations will result in no change in the resources allocated to a particular active stream.

Clients SHOULD assume the server is actively performing such pruning and SHOULD NOT declare a dependency on a stream it knows to have been closed.

5.4. Server Push

HTTP/3 server push is similar to what is described in HTTP/2 [RFC7540], but uses different mechanisms.

Each server push is identified by a unique Push ID. This Push ID is used in a single PUSH_PROMISE frame (see Section 4.2.6) which carries the request headers, possibly included in one or more DUPLICATE_PUSH frames (see Section 4.2.9), then included with the push stream which ultimately fulfills those promises.

Server push is only enabled on a connection when a client sends a MAX_PUSH_ID frame (see Section 4.2.8). A server cannot use server
push until it receives a MAX_PUSH_ID frame. A client sends additional MAX_PUSH_ID frames to control the number of pushes that a server can promise. A server SHOULD use Push IDs sequentially, starting at 0. A client MUST treat receipt of a push stream with a Push ID that is greater than the maximum Push ID as a connection error of type HTTP_LIMIT_EXCEEDED.

The header of the request message is carried by a PUSH_PROMISE frame (see Section 4.2.6) on the request stream which generated the push. This allows the server push to be associated with a client request. Ordering of a PUSH_PROMISE in relation to certain parts of the response is important (see Section 8.2.1 of [RFC7540]). Promised requests MUST conform to the requirements in Section 8.2 of [RFC7540].

The same server push can be associated with additional client requests using a DUPLICATE_PUSH frame (see Section 4.2.9). Ordering of a DUPLICATE_PUSH in relation to certain parts of the response is similarly important.

When a server later fulfills a promise, the server push response is conveyed on a push stream (see Section 3.2.2). The push stream identifies the Push ID of the promise that it fulfills, then contains a response to the promised request using the same format described for responses in Section 5.1.

Due to reordering, DUPLICATE_PUSH frames or push stream data can arrive before the corresponding PUSH_PROMISE frame. When a client receives a DUPLICATE_PUSH frame for an as-yet-unknown Push ID, the request headers of the push are not immediately available. The client can either delay generating new requests for content referenced following the DUPLICATE_PUSH frame until the request headers become available, or can initiate requests for discovered resources and cancel the requests if the requested resource is already being pushed. When a client receives a new push stream with an as-yet-unknown Push ID, both the associated client request and the pushed request headers 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.

If a promised server push is not needed by the client, the client SHOULD send a CANCEL_PUSH frame. If the push stream is already open or opens after sending the CANCEL_PUSH frame, a QUIC STOP_SENDING frame with an appropriate error code can also be used (e.g., HTTP_PUSH_REFUSED, HTTP_PUSH_ALREADY_IN_CACHE; see Section 8). This asks the server not to transfer additional data and indicates that it will be discarded upon receipt.
6. 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.

6.1. Idle Connections

Each QUIC endpoint declares an idle timeout during the handshake. If the 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 connection for new requests if the existing connection has been idle for longer than the server’s advertised idle timeout, and SHOULD do so if approaching the idle timeout.

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 19.2 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.

6.2. Connection Shutdown

Even when a connection is not idle, either endpoint can decide to stop using the connection and let the connection close gracefully. Since clients drive request generation, clients perform a connection shutdown by not sending additional requests on the connection; responses and pushed responses associated to previous requests will continue to completion. Servers perform the same function by communicating with clients.

Servers initiate the shutdown of a connection by sending a GOAWAY frame (Section 4.2.7). The GOAWAY frame indicates that client-initiated requests on lower stream IDs were or might be processed in this connection, while requests on the indicated stream ID and greater were rejected. This enables client and server to agree on which requests were accepted prior to the connection shutdown. This identifier MAY be lower than the stream limit identified by a QUIC MAX_STREAM_ID frame, and MAY be zero if no requests were processed. Servers SHOULD NOT increase the QUIC MAX_STREAM_ID limit after sending a GOAWAY frame.
Clients MUST NOT send new requests on the connection after receiving GOAWAY; a new connection MAY be established to send additional requests.

Some requests might already be in transit. If the client has already sent requests on streams with a Stream ID greater than or equal to that indicated in the GOAWAY frame, those requests will not be processed and MAY be retried by the client on a different connection. The client MAY cancel these requests. It is RECOMMENDED that the server explicitly reject such requests (see Section 5.1.2) in order to clean up transport state for the affected streams.

Requests on Stream IDs less than the Stream ID in the GOAWAY frame might have been processed; their status cannot be known until a response is received, the stream is reset individually, or the connection terminates. Servers MAY reject individual requests on streams below the indicated ID if these requests were not processed.

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.

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. A server MAY send multiple GOAWAY frames indicating different stream IDs, but MUST NOT increase the value they send in the last Stream ID, since clients might already have retried unprocessed requests on another connection. A server that is attempting to gracefully shut down a connection SHOULD send an initial GOAWAY frame with the last Stream ID set to the current value of QUIC’s MAX_STREAM_ID and SHOULD NOT increase the MAX_STREAM_ID thereafter. This signals to the client that a shutdown is imminent and that initiating further requests is prohibited. After allowing time for any in-flight requests (at least one round-trip time), the server MAY send another GOAWAY frame with an updated last Stream ID. This ensures that a connection can be cleanly shut down without losing requests.

Once all accepted requests have been processed, the server 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 HTTP_NO_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.

6.3. 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; the error code in this frame indicates to the peer why the connection is being closed. See Section 8 for error codes which can be used when closing a connection.

Before closing the connection, a GOAWAY 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.

6.4. 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 which interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST assume that any request which was sent, whether in whole or in part, might have been processed.

7. 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 header fields.

Extensions are permitted to use new frame types (Section 4.2), new settings (Section 4.2.5.1), new error codes (Section 8), or new unidirectional stream types (Section 3.2). Registries are established for managing these extension points: frame types (Section 10.3), settings (Section 10.4), error codes (Section 10.5), and stream types (Section 10.6).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames
Internet-Draft HTTP/3 April 2019

and 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.

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. In this case, it could also be necessary to coordinate when the revised layout comes into effect.

This document doesn’t mandate a specific method for negotiating the use of an extension but notes that a setting (Section 4.2.5.1) 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.

8. Error Handling

QUIC allows the application to abruptly terminate (reset) individual streams or the entire connection when an error is encountered. These are referred to as "stream errors" or "connection errors" and are described in more detail in [QUIC-TRANSPORT]. An endpoint MAY choose to treat a stream error as a connection error.

This section describes HTTP/3-specific error codes which can be used to express the cause of a connection or stream error.

8.1. HTTP/3 Error Codes

The following error codes are defined for use in QUIC RESET_STREAM frames, STOP_SENDING frames, and CONNECTION_CLOSE frames when using HTTP/3.

HTTP_NO_ERROR (0x00): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

HTTP_WRONG_SETTING_DIRECTION (0x01): A client-only setting was sent by a server, or a server-only setting by a client.

HTTP_PUSH_REFUSED (0x02): The server has attempted to push content which the client will not accept on this connection.

HTTP_INTERNAL_ERROR (0x03): An internal error has occurred in the HTTP stack.
HTTP_PUSH_ALREADY_IN_CACHE (0x04): The server has attempted to push content which the client has cached.

HTTP_REQUEST_CANCELLED (0x05): The request or its response is cancelled.

HTTP_INCOMPLETE_REQUEST (0x06): The client’s stream terminated without containing a fully-formed request.

HTTP_CONNECT_ERROR (0x07): The connection established in response to a CONNECT request was reset or abnormally closed.

HTTP_EXCESSIVE_LOAD (0x08): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

HTTP_VERSION_FALLBACK (0x09): The requested operation cannot be served over HTTP/3. The peer should retry over HTTP/1.1.

HTTP_WRONG_STREAM (0x0A): A frame was received on a stream where it is not permitted.

HTTP_LIMIT_EXCEEDED (0x0B): A Stream ID, Push ID, or Placeholder ID greater than the current maximum for that identifier was referenced.

HTTP_DUPLICATE_PUSH (0x0C): A Push ID was referenced in two different stream headers.

HTTP_UNKNOWN_STREAM_TYPE (0x0D): A unidirectional stream header contained an unknown stream type.

HTTP_WRONG_STREAM_COUNT (0x0E): A unidirectional stream type was used more times than is permitted by that type.

HTTP_CLOSED_CRITICAL_STREAM (0x0F): A stream required by the connection was closed or reset.

HTTP_WRONG_STREAM_DIRECTION (0x0010): A unidirectional stream type was used by a peer which is not permitted to do so.

HTTP_EARLY_RESPONSE (0x0011): The remainder of the client’s request is not needed to produce a response. For use in STOP_SENDING only.

HTTP_MISSING_SETTINGS (0x0012): No SETTINGS frame was received at the beginning of the control stream.
HTTP_UNEXPECTED_FRAME (0x0013): A frame was received which was not permitted in the current state.

HTTP_REQUEST_REJECTED (0x0014): A server rejected a request without performing any application processing.

HTTP_GENERAL_PROTOCOL_ERROR (0x00FF): Peer violated protocol requirements in a way which doesn’t match a more specific error code, or endpoint declines to use the more specific error code.

HTTP_MALFORMED_FRAME (0x01XX): An error in a specific frame type. If the frame type is "0xfe" or less, the type is included as the last byte of the error code. For example, an error in a MAX_PUSH_ID frame would be indicated with the code (0x10D). The last byte "0xff" is used to indicate any frame type greater than "0xfe".

9. Security Considerations

The security considerations of HTTP/3 should be comparable to those of HTTP/2 with TLS. Note that where HTTP/2 employs PADDING frames and Padding fields in other frames to make a connection more resistant to traffic analysis, HTTP/3 can rely on QUIC PADDING frames or employ the reserved frame and stream types discussed in Section 4.2.10 and Section 3.2.3.

When HTTP Alternative Services is used for discovery for HTTP/3 endpoints, the security considerations of [ALTSVC] also apply.

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.

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.

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. IANA Considerations

10.1. 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

10.2. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter" registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 2.2.1

10.3. 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 space is split into three spaces that are governed by different policies. Values between "0x00" and "0x3f" (in hexadecimal) are assigned via the Standards Action or IESG Review policies [RFC8126]. Values from "0x40" to "0x3fff" operate on the Specification Required policy [RFC8126]. All other values are assigned to Private Use [RFC8126].

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [RFC7540], 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.

New entries in this registry require the following information:

Frame Type: A name or label for the frame type.

Code: The 62-bit code assigned to the frame type.
Specification: A reference to a specification that includes a description of the frame layout and its semantics, including any parts of the frame that are conditionally present.

The entries in the following table are registered by this document.

<table>
<thead>
<tr>
<th>Frame Type</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>0x0</td>
<td>Section 4.2.1</td>
</tr>
<tr>
<td>HEADERS</td>
<td>0x1</td>
<td>Section 4.2.2</td>
</tr>
<tr>
<td>PRIORITY</td>
<td>0x2</td>
<td>Section 4.2.3</td>
</tr>
<tr>
<td>CANCEL_PUSH</td>
<td>0x3</td>
<td>Section 4.2.4</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>0x4</td>
<td>Section 4.2.5</td>
</tr>
<tr>
<td>PUSH_PROMISE</td>
<td>0x5</td>
<td>Section 4.2.6</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x6</td>
<td>N/A</td>
</tr>
<tr>
<td>GOAWAY</td>
<td>0x7</td>
<td>Section 4.2.7</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x8</td>
<td>N/A</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x9</td>
<td>N/A</td>
</tr>
<tr>
<td>MAX_PUSH_ID</td>
<td>0xD</td>
<td>Section 4.2.8</td>
</tr>
<tr>
<td>DUPLICATE_PUSH</td>
<td>0xE</td>
<td>Section 4.2.9</td>
</tr>
</tbody>
</table>

Additionally, each code of the format "0x1f * N + 0x21" for integer values of N (that is, "0x21", "0x40", ..., through "0x3FFFFFFF87FFFFFE") MUST NOT be assigned by IANA.

10.4. Settings Parameters

This document establishes a registry for HTTP/3 settings. The "HTTP/3 Settings" registry governs a 62-bit space. This space is split into three spaces that are governed by different policies. Values between "0x00" and "0x3f" (in hexadecimal) are assigned via the Standards Action or IESG Review policies [RFC8126]. Values from "0x40" to "0x3fff" operate on the Specification Required policy [RFC8126]. All other values are assigned to Private Use [RFC8126].
The designated experts are the same as those for the "HTTP/2 Settings" registry defined in [RFC7540].

While this registry is separate from the "HTTP/2 Settings" registry defined in [RFC7540], 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.

New registrations are advised to provide the following information:

Name: A symbolic name for the setting. Specifying a setting name is optional.

Code: The 62-bit code assigned to the setting.

Specification: An optional reference to a specification that describes the use of the setting.

The entries in the following table are registered by this document.

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>0x2</td>
<td>N/A</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x3</td>
<td>N/A</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x4</td>
<td>N/A</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x5</td>
<td>N/A</td>
</tr>
<tr>
<td>MAX_HEADER_LIST_SIZE</td>
<td>0x6</td>
<td>Section 4.2.5.1</td>
</tr>
<tr>
<td>NUM_PLACEHOLDERS</td>
<td>0x9</td>
<td>Section 4.2.5.1</td>
</tr>
</tbody>
</table>

Additionally, each code of the format "0x1f * N + 0x21" for integer values of N (that is, "0x21", "0x40", ..., through "0x3FFFFFFFFFFFFFFFE") MUST NOT be assigned by IANA.

10.5. Error Codes

This document establishes a registry for HTTP/3 error codes. The "HTTP/3 Error Code" registry manages a 16-bit space. The "HTTP/3 Error Code" registry operates under the "Expert Review" policy [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.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is optional.

Code: The 16-bit error code value.

Description: A brief description of the error code semantics, longer if no detailed specification is provided.

Specification: An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_NO_ERROR</td>
<td>0x0000</td>
<td>No error</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_WRONG_SETTING_DIRECTION</td>
<td>0x0001</td>
<td>Setting sent in wrong direction</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_PUSH_REFUSED</td>
<td>0x0002</td>
<td>Client refused pushed content</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_INTERNAL_ERROR</td>
<td>0x0003</td>
<td>Internal error</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_PUSH_ALREADY_IN_CACHE</td>
<td>0x0004</td>
<td>Pushed content already cached</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_REQUEST_CANCELLED</td>
<td>0x0005</td>
<td>Data no longer needed</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_INCOMPLETE_REQUEST</td>
<td>0x0006</td>
<td>Stream terminated</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>Error Code</td>
<td>Value</td>
<td>Message Description</td>
<td>Section</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>HTTP_CONNECT_ERROR</td>
<td>0x0007</td>
<td>TCP reset or error on CONNECT request</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_EXCESSIVE_LOAD</td>
<td>0x0008</td>
<td>Peer generating excessive load</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_VERSION_FALLBACK</td>
<td>0x0009</td>
<td>Retry over HTTP/1.1</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_WRONG_STREAM</td>
<td>0x000A</td>
<td>A frame was sent on the wrong stream</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_LIMIT_EXCEEDED</td>
<td>0x000B</td>
<td>An identifier limit was exceeded</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_DUPLICATE_PUSH</td>
<td>0x000C</td>
<td>Push ID was fulfilled multiple times</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_UNKNOWN_STREAM_TYPE</td>
<td>0x000D</td>
<td>Unknown unidirectional stream type</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_WRONG_STREAM_COUNT</td>
<td>0x000E</td>
<td>Too many unidirectional streams</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_CLOSED_CRITICAL_STREAM</td>
<td>0x000F</td>
<td>Critical stream was closed</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_WRONG_STREAM_DIRECTION</td>
<td>0x0010</td>
<td>Unidirectional stream in wrong direction</td>
<td>8.1</td>
</tr>
<tr>
<td>HTTP_EARLY_RESPONSE</td>
<td>0x0011</td>
<td>Remainder of request not needed</td>
<td>8.1</td>
</tr>
</tbody>
</table>
### 10.6. 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 space is split into three spaces that are governed by different policies. Values between "0x00" and 0x3f (in hexadecimal) are assigned via the Standards Action or IESG Review policies [RFC8126]. Values from "0x40" to "0x3fff" operate on the Specification Required policy [RFC8126]. All other values are assigned to Private Use [RFC8126].

New entries in this registry require the following information:

- **Stream Type**: A name or label for the stream type.
- **Code**: The 62-bit code assigned to the stream type.
- **Specification**: A reference to a specification that includes a description of the stream type, including the layout semantics of its payload.
- **Sender**: Which endpoint on a connection may initiate a stream of this type. Values are "Client", "Server", or "Both".

The entries in the following table are registered by this document.

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Code</th>
<th>Description</th>
<th>Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_MISSING_SETTINGS</td>
<td>0x0012</td>
<td>No SETTINGS frame received</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_UNEXPECTED_FRAME</td>
<td>0x0013</td>
<td>Frame not permitted in the current state</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_REQUEST_REJECTED</td>
<td>0x0014</td>
<td>Request not processed</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>HTTP_MALFORMED_FRAME</td>
<td>0x01XX</td>
<td>Error in frame formatting</td>
<td>Section 8.1</td>
</tr>
</tbody>
</table>
Additionally, each code of the format "0x1f * N + 0x21" for integer values of N (that is, "0x21", "0x40", ..., through "0x3FFFFFFFFFFFFFFE") MUST NOT be assigned by IANA.

11. References

11.1. Normative References


11.2. Informative References


11.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

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 primarily where necessary to accommodate the differences in behavior between QUIC and TCP (lack of ordering, support for streams). We intend to avoid gratuitous changes which make it difficult or impossible to build extensions with the same semantics applicable to both protocols at once.

These 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 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.

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 [RFC7540]. However, QUIC includes many features (e.g., flow control) which 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 equivalent frames between the two mappings are not identical.

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.

For example, implicit in the HTTP/2 prioritization scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of priority changes in HTTP/3, PRIORITY frames are sent as the first frame on a request stream or on the control stream and exclusive prioritization has been removed. HTTP/3 permits the prioritisation of requests, pushes and placeholders that each exist in separate identifier spaces. The HTTP/3 PRIORITY frame replaces the stream dependency field with fields that can identify the element of interest and its dependency.

Likewise, HPACK was designed with the assumption of in-order delivery. A sequence of encoded header blocks 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. As a result, HTTP/3 uses a modified version of HPACK, described in [QPACK].

Frame type definitions in HTTP/3 often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allow for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/3 use an identifier rather than a Stream ID (e.g. Push IDs in PRIORITY frames). 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 which depend on the presence of flags need to allocate space for flags as part of their frame payload.
Other than this issue, 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 would be portable to HTTP/2 in the same manner.

Below is a listing of how each HTTP/2 frame type is mapped:

**DATA (0x0):** Padding is not defined in HTTP/3 frames. See Section 4.2.1.

**HEADERS (0x1):** The PRIORITY region of HEADERS is not defined in HTTP/3 frames. A separate PRIORITY frame is used in all cases. Padding is not defined in HTTP/3 frames. See Section 4.2.2.

**PRIORITY (0x2):** As described above, the PRIORITY frame references a variety of identifiers. It is sent as the first frame on a request streams or on the control stream. See Section 4.2.3.

**RST_STREAM (0x3):** RST_STREAM frames do not exist, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL_PUSH frame (Section 4.2.4).

**SETTINGS (0x4):** SETTINGS frames are sent only at the beginning of the connection. See Section 4.2.5 and Appendix A.3.

**PUSH_PROMISE (0x5):** The PUSH_PROMISE does not reference a stream; instead the push stream references the PUSH_PROMISE frame using a Push ID. See Section 4.2.6.

**PING (0x6):** PING frames do not exist, since QUIC provides equivalent functionality.

**GOAWAY (0x7):** GOAWAY is sent only from server to client and does not contain an error code. See Section 4.2.7.

**WINDOW_UPDATE (0x8):** WINDOW_UPDATE frames do not exist, since QUIC provides flow control.

**CONTINUATION (0x9):** CONTINUATION frames do not exist; 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 [RFC7540] 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 10.3.
A.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, at the beginning of the connection, 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 options that are retained in HTTP/3 have the same value as in HTTP/2.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

**SETTINGS_HEADER_TABLE_SIZE:** See [QPACK].

**SETTINGS_ENABLE_PUSH:** This is removed in favor of the MAX_PUSH_ID which provides a more granular control over server push.

**SETTINGS_MAX_CONCURRENT_STREAMS:** QUIC controls the largest open Stream ID as part of its flow control logic. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

**SETTINGS_INITIAL_WINDOW_SIZE:** QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

**SETTINGS_MAX_FRAME_SIZE:** This setting has no equivalent in HTTP/3. Specifying it in the SETTINGS frame is an error.

**SETTINGS_MAX_HEADER_LIST_SIZE:** See Section 4.2.5.1.

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 which use the full 32-bit space. Settings ported from HTTP/2 might choose to redefine the format of their settings to avoid using the 62-bit encoding.

Settings need to be defined separately for HTTP/2 and HTTP/3. The IDs of settings defined in [RFC7540] 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 10.4.
A.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, there is no direct portability of HTTP/2 error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to the HTTP/3 error codes as follows:

NO_ERROR (0x0): HTTP_NO_ERROR in Section 8.1.

PROTOCOL_ERROR (0x1): No single mapping. See new HTTP_MALFORMED_FRAME error codes defined in Section 8.1.

INTERNAL_ERROR (0x2): HTTP_INTERNAL_ERROR in Section 8.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA from the QUIC layer.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION from the QUIC layer.

FRAME_SIZE_ERROR (0x6): HTTP_MALFORMED_FRAME error codes defined in Section 8.1.

REFUSED_STREAM (0x7): HTTP_REQUEST_REJECTED (in Section 8.1) is used to indicate that a request was not processed. Otherwise, not applicable because QUIC handles stream management. A STREAM_ID_ERROR at the QUIC layer is used for streams that are improperly opened.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 8.1.

COMPRESSION_ERROR (0x9): Multiple error codes are defined in [QPACK].

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 8.1.

ENHANCE_YOUR_CALM (0xb): HTTP_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): HTTP_VERSION_FALLBACK in Section 8.1.

Error codes need to be defined for HTTP/2 and HTTP/3 separately. See Section 10.5.

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-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.2. 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.3. 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.4. Since draft-ietf-quic-http-16
   o Rename "HTTP/QUIC" to "HTTP/3" (#1973)
   o Changes to PRIORITY frame (#1865, #2075)
     * Permitted as first frame of request streams
     * Remove exclusive reprioritization
     * Changes to Prioritized Element Type bits
   o Define DUPLICATE_PUSH frame to refer to another PUSH_PROMISE (#2072)
   o Set defaults for settings, allow request before receiving SETTINGS (#1809, #1846, #2038)
   o Clarify message processing rules for streams that aren’t closed (#1972, #2003)
   o Removed reservation of error code 0 and moved HTTP_NO_ERROR to this value (#1922)
   o Removed prohibition of zero-length DATA frames (#2098)

B.5. Since draft-ietf-quic-http-15

   Substantial editorial reorganization; no technical changes.


   o Recommend sensible values for QUIC transport parameters (#1720,#1806)
   o Define error for missing SETTINGS frame (#1697,#1808)
   o Setting values are variable-length integers (#1556,#1807) and do not have separate maximum values (#1820)
   o Expanded discussion of connection closure (#1599,#1717,#1712)
   o HTTP_VERSION_FALLBACK falls back to HTTP/1.1 (#1677,#1685)
B.7. 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.8. 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)

   - Moved QPACK table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

B.10. Since draft-ietf-quic-http-10
   - Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.11. 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.12. Since draft-ietf-quic-http-08
- Clarified connection coalescing rules (#940, #1024)

B.13. 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.14. Since draft-ietf-quic-http-06
- Track changes in QUIC error code usage (#485)

B.15. 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)

- 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.17. Since draft-ietf-quic-http-03

None.

B.18. Since draft-ietf-quic-http-02

- Track changes in transport draft


- 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.20. 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)
o Updated ALPN token and Alt-Svc guidance (#13,#87)

o Application-layer-defined error codes (#19,#74)

B.21. Since draft-shade-quic-http2-mapping-00

o Adopted as base for draft-ietf-quic-http

o Updated authors/editors list

Acknowledgements

The original authors of this specification were Robbie Shade and Mike Warres.

A substantial portion of Mike’s contribution was supported by Microsoft during his employment there.

Author’s Address

Mike Bishop (editor)
Akamai

Email: mbishop@evequefou.be
Using HTTP/2 as a Transport for Arbitrary Bytestreams
draft-kinnear-httpbis-http2-transport-00

Abstract

HTTP/2 provides multiplexing of HTTP requests over a single underlying transport connection. HTTP/2 Transport defines a transport abstraction provided by HTTP/2 framing that is separate from HTTP semantics.

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 March 14, 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.
1. Introduction

HTTP/2 [RFC7540] provides a framing layer that describes the exchange of HTTP messages following HTTP semantics. This framing layer provides multiplexing of multiple streams on a single underlying transport connection, flow control, stream dependencies and priorities, and exchange of configuration information between endpoints. HTTP/2 also defines the mapping of HTTP semantics onto that framing layer.

This document defines the use of the HTTP/2 framing layer as a transport for arbitrary byte streams without the use of HTTP semantics.

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] when, and only when, they appear in all capitals, as shown here.

2. The STREAM HTTP/2 Frame

This document defines a new HTTP/2 frame type called STREAM, that allows endpoints to open HTTP/2 streams without header values. Either endpoint can send this frame to open a stream. STREAM frames are treated in all ways as HEADERS frames, including in the stream state machine, but are not required to contain any header values.
2.1. Syntax

The STREAM frame type is 0xd (decimal 13) and contains similar fields to that of the HEADERS frame.

A STREAM frame is shown below.

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+                  +---------------+
|Pad Length? (8)|                  |Stream Dependency? (31) |
|E|                 +---------------+                  |
| Weight? (8) |                  | E | Stream Dependency? (31) |
|+---------------+                  |+---------------+                  |
|                           +---------------+                  |
+---------------------------------------------------------------+
```

Figure 1: STREAM Frame Format

The STREAM frame contains the following fields:

Pad Length: An 8-bit field containing the length of the frame padding in units of octets. This field is only present if the PADDED flag is set.

E: A single-bit flag indicating that the stream dependency is exclusive (see Section 5.3 of [RFC7540]). This field is only present if the PRIORITY flag is set.

Stream Dependency: A 31-bit stream identifier for the stream that this stream depends on (see Section 5.3 of [RFC7540]). This field is only present if the PRIORITY flag is set.

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see Section 5.3 of [RFC7540]). Add one to the value to obtain a weight between 1 and 256. This field is only present if the PRIORITY flag is set.

Padding: Padding octets.

The STREAM frame defines the following flags:

PADDED (0x8): When set, bit 3 indicates that the Pad Length field and any padding that it describes are present.
PRIORITY (0x20): When set, bit 5 indicates that the Exclusive Flag (E), Stream Dependency, and Weight fields are present; see Section 5.3 of [RFC7540].

STREAM frames MUST be associated with a stream. If a STREAM frame is received whose stream identifier field is 0x0, the recipient MUST respond with a connection error of type PROTOCOL_ERROR.

The STREAM frame can include padding. Padding fields and flags are identical to those defined for DATA frames.

Prioritization information in a STREAM frame is logically equivalent to a separate PRIORITY frame, but inclusion in a STREAM frame avoids the potential for churn in stream prioritization when new streams are created. Prioritization fields in STREAM frames subsequent to the first on a stream reprioritize the stream (Section 5.3.3 of [RFC7540]).

2.2. Advertising Support for STREAM Frames

As defined in [RFC7540], both endpoints can send SETTINGS_MAX_CONCURRENT_STREAMS in SETTINGS frames to indicate the number of streams that the sender permits the receiver to create. This limit applies to streams created via the STREAM frame as well as streams created via HEADERS frames.

2.3. Processing STREAM Frames

The STREAM 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 a client that implements support, the STREAM frame behaves in the same manner as a HEADERS frame, but does not carry any header blocks. This changes the connection state in the same manner as a HEADERS frame, described in Section 4.3 of [RFC7540].

STREAM frames can be sent on a stream in the "idle", "reserved (local)", "open", or "half-closed (remote)" state. STREAM frames can be sent by either endpoint on a connection.

Streams created via a STREAM frame are multiplexed in the same manner on the underlying transport connection as streams created via a HEADERS frame. Flow control also applies to these streams in the same way. Flow control, stream dependencies, and priorities continue to apply to streams as defined by [RFC7540].

Anywhere an endpoint would be permitted to send a HEADERS frame by [RFC7540], it is likewise permitted to send a STREAM frame.
A stream is closed via a RST_STREAM frame or by setting the END_STREAM flag on a DATA frame.

3. IANA Considerations

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

- Frame Type: STREAM
- Code: 0xd
- Specification: [[RFC Editor: Please fill in this value with the RFC number for this document.]]

4. Security Considerations

5. Acknowledgments

Thanks to Anthony Chivetta, Joshua Otto, and Valentin Pistol for their contributions in the design and prototyping of this work.

6. Normative References


Authors' Addresses

Eric Kinnear
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: ekinnear@apple.com
Internet-Draft                HTTP/2 Transport                September 2018

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com