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.
Copyright Notice

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

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

Table of Contents

1. Introduction ................................................. 3
   1.1. 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
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
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>Allows changing the request method from POST to GET</th>
<th>Permanent</th>
<th>Temporary</th>
</tr>
</thead>
<tbody>
<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]

[RFC2119]

[RFC2818]
7.2. Informative References


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/
The Cache HTTP Response Header
draft-ietf-httpbis-cache-header-00

Abstract

To aid debugging, HTTP caches often append headers to a response
detailing how they handled the request. This specification codifies
that practice and updates it for HTTP’s current caching model.

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.org/ [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 July 31, 2019.
1. Introduction

To aid debugging, HTTP caches often append headers to a response detailing how they handled the request.

Unfortunately, the semantics of these headers are often unclear, and both the semantics and syntax used vary greatly between implementations.

This specification defines a single, new HTTP response header field, "Cache" for this purpose.

For example:

Cache: HIT_FRESH; node="reverse-proxy.example.com:80";
        key="https://example.com/foo|Accept-Encoding:gzip",
Cache: HIT_STALE; node="FooCDN parent"; fresh=-45; age=200; latency=3,
Cache: MISS; node="FooCDN edge"; fresh=-45; age=200; latency=98
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 ABNF as defined in [RFC5234], along with the "%s" extension for case sensitivity defined in [RFC7405].

2. The Cache HTTP Response Header

The Cache HTTP response header indicates the handling of the request corresponding to the response it occurs within by caches along the path.

Its value is a Parameterised List [I-D.ietf-httpbis-header-structure]:

    Cache   = sh-param-list

Each member of the parameterised list represents a cache that has handled the request.

The first member of the list represents the cache closest to the origin server, and the last member of the list represents the cache closest to the user agent (possibly including the user agent’s cache itself, if it chooses to append a value).

Caches determine when it is appropriate to add the Cache header field to a response. Some might decide to add it to all responses, whereas others might only do so when specifically configured to, or when the request contains a header that activates a debugging mode.

When adding a value to the Cache header field, caches SHOULD preserve the existing contents of the header, to allow debugging of the entire chain of caches handling the request.

Identifiers in the parameterised list members are expected to be cache-actions:
cache-action = %s"HIT_FRESH"
   / %s"HIT_STALE"
   / %s"HIT_REFRESH_MODIFIED"
   / %s"HIT_REFRESH_NOT_MODIFIED"
   / %s"HIT_REFRESH_STALE"
   / %s"MISS"
   / %s"MISS_CLIENT"
   / %s"BYPASS"
   / %s"ERROR"

The semantics of cache-actions are:

- HIT_FRESH - The cache used a fresh stored response to satisfy the request without going forward.
- HIT_STALE - The cache used a stale stored response to satisfy the request without going forward.
- HIT_REFRESH_MODIFIED - The cache had a stale stored response, went forward to validate it, and used the new response to satisfy the request.
- HIT_REFRESH_NOT_MODIFIED - The cache had a stale stored response, went forward to validate it, and used the stored response to satisfy the request.
- HIT_REFRESH_STALE - The cache had a stale stored response, went forward to validate it, and encountered a problem, so the stored response was used to satisfy the request.
- MISS - The cache did not have a stored response, so the request was forwarded.
- MISS_CLIENT - The client included request directives (e.g., Pragma, Cache-Control) that prevented the cache from returning a response, so the request was forwarded.
- BYPASS - The cache was configured to forward the request without attempting to use a stored response.
- ERROR - The cache was unable to use a stored response or obtain one by going forward.

Caches SHOULD use the most specific cache-action to a given response, but are not required to use all cache-actions. Future updates to this specification can add additional cache-actions.
Each member of the Cache header can also have any (or all, or none) of the following parameters:

node = sh-string
fresh = sh-integer
age = sh-integer
cacheable = sh-boolean
key = sh-string
latency = sh-integer
cl_nm = sh-boolean

Their semantics are:

- "node" - a string identifying for the cache node. MAY be a hostname, IP address, or alias.
- "fresh" - an integer indicating the cache’s estimation of the freshness lifetime ([RFC7234], Section 4.2.1) of this response in seconds, including any locally applied configuration. MAY be negative.
- "age" - an integer indicating the cache’s estimation of the age ([RFC7234], Section 4.2.3) of this response in seconds. MUST be 0 or greater.
- "cacheable" - a boolean indicating whether the cache can store this response, according to [RFC7234], Section 3 and any locally applied configuration.
- "key" - a string representing the key that the cache has associated with this response. This might include the request URL, request headers, and other values.
- "latency" - an integer indicating the amount of time in milliseconds between the receipt of a complete set of request headers and sending the complete set of response headers of this response, from the viewpoint of the cache. Note that this may not include buffering time in transport protocols and similar delays.
- "cl_nm" - a boolean indicating whether the response to the client had a 304 Not Modified status code.

While all of these parameters are OPTIONAL, caches are encouraged to use the ‘node’ parameter to identify themselves.
3. Security Considerations

Information about a cache’s content can be used to infer the activity of those using it. Generally, access to sensitive information in a cache is limited to those who are authorised to access that information (using a variety of techniques), so this does not represent an attack vector in the general sense.

However, if the Cache header is exposed to parties who are not authorised to obtain the response it occurs within, it could expose information about that data.

For example, if an attacker were able to obtain the Cache header from a response containing sensitive information and access were limited to one person (or limited set of people), they could determine whether that information had been accessed before. This is similar to the information exposed by various timing attacks, but is arguably more reliable, since the cache is directly reporting its state.

Mitigations include use of encryption (e.g., TLS [RFC8446])) to protect the response, and careful controls over access to response headers (as are present in the Web platform). When in doubt, the Cache header field can be omitted.

4. References

4.1. Normative References

[I-D.ietf-httpbis-header-structure]


4.2. Informative References


4.3. URIs

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

Author’s Address

Mark Nottingham
Fastly

Email: mnot@mnot.net
URI: https://www.mnot.net/
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 https://github.com/httpwg/http-extensions/labels/client-hints.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at 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."
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 ......................................... 8
   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

- 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/
Structured Headers for HTTP
draft-ietf-httpbis-header-structure-10

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.
Internet-Draft
3.

Structured Headers for HTTP

April 2019

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
member-name
member-value
key
lcalpha

=
=
=
=
=
=

dict-member *( OWS "," OWS dict-member )
member-name "=" member-value
key
sh-item
lcalpha *( lcalpha / DIGIT / "_" / "-" )
%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,
individual
optional.
unless the

a header field specification will define the semantics of
keys, as well as whether their presence is required or
Recipients MUST ignore keys that are undefined or unknown,
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

Nottingham & Kamp

Expires October 19, 2019

[Page 7]


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 inner-list 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:
sh-param-list = param-item *( OWS "," OWS param-item )
param-item = primary-id *parameter
primary-id = sh-token
parameter = OWS ";" OWS param-name [ "=" param-value ]
param-name = key
param-value = sh-item

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:

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:

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:
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 serialize 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 structure 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 ({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].
5. Append DQUOTE to output.
6. 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 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
referring to 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 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:

Nottingham & Kamp Expires October 19, 2019 [Page 20]
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-%x1f 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, ";", ",", ":", "/", ";+", ";*" or ";/":

      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


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
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).
o Clamp integers to 15 digits (#737).

C.2. Since draft-ietf-httpbis-header-structure-08

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

C.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

   o Throw an error on trailing garbage.

       o Replaced with draft-nottingham-structured-headers.

C.10. Since draft-ietf-httpbis-header-structure-00
       o Added signed 64bit integer type.

       o Drop UTF8, and settle on BCP137 ::EmbeddedUnicodeChar for h1-
         unicode-string.

       o 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
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 https://github.com/httpwg/http-extensions/labels/secondary-certs [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 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
1. Introduction

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                                      Server
-- (HTTP) GET /protected ------------------->
<----------------- (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 use 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:

\[(Exporter \& 0x3fffffff) | 0x80000000\]

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
Internet-Draft        Secondary Cert Auth in HTTP/2           April 2019

[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.

```
+---------------------------------------------------------------+
|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:

```
+---------------------------------------------------------------+
|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 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".

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
+-------------------------------+-------------------------------+
|        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).

id-ce-requiredDomain OBJECT IDENTIFIER ::=  { id-ce TBD1 }

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]

8.2. 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:

- 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
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.
Internet-Draft        HTTP Representation Variants            March 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
2.  The "Variants" HTTP Header Field . . . . . . . . . . . . . .   4
   2.1.  Relationship to Vary . . . . . . . . . . . . . . . . . .   7
3.  The "Variant-Key" HTTP Header Field . . . . . . . . . . . . .   8
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.

Nottingham             Expires September 26, 2019               [Page 4]
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;brotli
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;fr;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"]  // prefers German; will not accept English
```

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:

```
[ "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
Variants: Accept-Encoding;br;gzip
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


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/
Hypertext Transfer Protocol Version 3 (HTTP/3)
draft-ietf-quic-http-20

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
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]) 0x1badaba, it could specify the following header field:

```
Alt-Svc: h3=":49288";quic="1,,1badaba,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.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           0x01 (i)                          ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Push ID (i)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: Push Stream Header

Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type 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 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>
<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 2: Prioritized Element 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>

Table 3: Element Dependency Types

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)                           ...\n+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
\                                 Value (i)                                ...\n+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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.

```
 0                   1                   2                   3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Stream ID (i)                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

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.

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.
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.

```
+--------+  +--------+  +--------+
|   P    |  |   P    |  |   P    |
| / \    |  | / \    |  | / \    |
| I I    |  | I I    |  | I 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
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 "0x3FFFFFFFFFFFFFFE") 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 "0x3FFFFFFFFFFFFFFE") 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>Code</td>
<td>Description</td>
<td>Section</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>HTTP_CONNECT_ERROR 0x0007</td>
<td>TCP reset or error on CONNECT request</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_EXCESSIVE_LOAD 0x0008</td>
<td>Peer generating excessive load</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_VERSION_FALLBACK 0x0009</td>
<td>Retry over HTTP/1.1</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_WRONG_STREAM 0x000A</td>
<td>A frame was sent on the wrong stream</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_LIMIT_EXCEEDED 0x000B</td>
<td>An identifier limit was exceeded</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_DUPLICATE_PUSH 0x000C</td>
<td>Push ID was fulfilled multiple times</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_UNKNOWN_STREAM_TYPE 0x000D</td>
<td>Unknown unidirectional stream type</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_WRONG_STREAM_COUNT 0x000E</td>
<td>Too many unidirectional streams</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_CLOSED_CRITICAL_STREAM 0x000F</td>
<td>Critical stream was closed</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_WRONG_STREAM_DIRECTION 0x0010</td>
<td>Unidirectional stream in wrong direction</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_EARLY_RESPONSE 0x0011</td>
<td>Remainder of request not needed</td>
<td>8.1</td>
<td></td>
</tr>
<tr>
<td>HTTP_MISSING_SETTINGS</td>
<td>0x0012</td>
<td>No SETTINGS frame received</td>
<td>Section 8.1</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------</td>
<td>----------------------------</td>
<td>-------------</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>

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.
### Stream Type | Code | Specification | Sender
---|---|---|---
Control Stream | 0x00 | Section 3.2.1 | Both
Push Stream | 0x01 | Section 5.4 | Server

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


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_INTERRUPTED 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

- Rename "HTTP/QUIC" to "HTTP/3" (#1973)
- Changes to PRIORITY frame (#1865, #2075)
  - Permitted as first frame of request streams
  - Remove exclusive reprioritization
  - Changes to Prioritized Element Type bits
- Define DUPLICATE_PUSH frame to refer to another PUSH_PROMISE (#2072)
- Set defaults for settings, allow request before receiving SETTINGS (#1809, #1846, #2038)
- Clarify message processing rules for streams that aren’t closed (#1972, #2003)
- Removed reservation of error code 0 and moved HTTP_NO_ERROR to this value (#1922)
- Removed prohibition of zero-length DATA frames (#2098)

B.5. Since draft-ietf-quic-http-15

Substantial editorial reorganization; no technical changes.


- Recommend sensible values for QUIC transport parameters (#1720, #1806)
- Define error for missing SETTINGS frame (#1697, #1808)
- Setting values are variable-length integers (#1556, #1807) and do not have separate maximum values (#1820)
- Expanded discussion of connection closure (#1599, #1717, #1712)
- HTTP_VERSION_FALLBACK falls back to HTTP/1.1 (#1677, #1685)
B.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
   o Clarified connection coalescing rules (#940, #1024)

B.13. Since draft-ietf-quic-http-07
   o Changes for integer encodings in QUIC (#595,#905)
   o Use unidirectional streams as appropriate (#515, #240, #281, #886)
   o Improvement to the description of GOAWAY (#604, #898)
   o Improve description of server push usage (#947, #950, #957)

B.14. Since draft-ietf-quic-http-06
   o Track changes in QUIC error code usage (#485)

B.15. Since draft-ietf-quic-http-05
   o Made push ID sequential, add MAX_PUSH_ID, remove SETTINGS_ENABLE_PUSH (#709)
   o Guidance about keep-alive and QUIC PINGs (#729)
   o Expanded text on GOAWAY and cancellation (#757)

   o Cite RFC 5234 (#404)
   o Return to a single stream per request (#245,#557)
   o Use separate frame type and settings registries from HTTP/2 (#81)
   o SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)
   o Restored GOAWAY (#696)
   o Identify server push using Push ID rather than a stream ID (#702,#281)
   o 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-01

Abstract

HTTP/2 provides multiplexing of HTTP requests over a single underlying transport connection. HTTP/2 Transport defines a transport abstraction enabling delivery of byte stream and datagram data using streams of an HTTP/2 connection.

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 11, 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.
HTTP/2 [RFC7540] provides a framing layer that describes the exchange of HTTP messages. This framing layer includes multiplexing of multiple streams on a single underlying transport connection, flow control, stream dependencies and priorities, and exchange of configuration information between endpoints.

Section 8.3 of [RFC7540] defines the HTTP CONNECT method for HTTP/2, which converts a HTTP/2 stream into a tunnel for arbitrary data. [RFC8441] describes the use of the extended CONNECT method to negotiate the use of the WebSocket Protocol [RFC6455] on an HTTP/2 stream.

This document defines protocol names for use in the extended CONNECT handshake that allow negotiation of HTTP/2 streams that transport arbitrary byte streams or datagrams. It also extends the CONNECT handshake to allow both endpoints of an HTTP/2 connection to establish streams that tunnel data. Being able to transport arbitrary data on individual HTTP/2 streams allows an underlying connection to be shared by multiple protocols and allows all protocols to benefit from the features provided by HTTP/2 framing.

1.1. Notational Conventions

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

As described in Section 5.5 of [RFC7540], SETTINGS parameters allow endpoints to negotiate use of protocol extensions that would otherwise generate protocol errors. Use of the CONNECT method extension defined in [RFC6455] requires the SETTINGS_ENABLE_CONNECT_PROTOCOL parameter to be received by a client prior to its use.

This document introduces another SETTINGS parameter, SETTINGS_ENABLE_BIDIRECTIONAL_CONNECT, which MUST have a value of 0 or 1.

Once a SETTINGS_ENABLE_BIDIRECTIONAL_CONNECT parameter has been sent with a value of 1, an endpoint MUST NOT send the parameter with a value of 0.

Upon receipt of SETTINGS_ENABLE_BIDIRECTIONAL_CONNECT with a value of 1, an endpoint MAY use the extended CONNECT defined in [RFC6455] with the protocol values defined in this document. An endpoint that supports receiving the extended CONNECT method SHOULD send this setting with a value of 1.

Note that [RFC6455] restricts SETTINGS_ENABLE_CONNECT_PROTOCOL to have no effect if received by a server. This document modifies that restriction and allows both SETTINGS_ENABLE_CONNECT_PROTOCOL and SETTINGS_ENABLE_BIDIRECTIONAL_CONNECT to take effect if received by either endpoint of an HTTP/2 connection.

3. Negotiating Byte Stream and Datagram Tunnels

[RFC6455] defines the psuedo-header field :protocol which can indicate the protocol intended to be used on the tunnel established by the CONNECT method. Values for the :protocol psuedo-header field are maintained in an Upgrade Token Registry established by [RFC7230] for protocol-name tokens.

After receiving both SETTINGS_ENABLE_CONNECT_PROTOCOL and SETTINGS_ENABLE_BIDIRECTIONAL_CONNECT, either endpoint of an HTTP/2 connection can send a request in HEADERS frames to establish a byte stream or datagram tunnel via the extended CONNECT method. Similarly, either endpoint may be required to respond to an incoming CONNECT request seeking to establish such a tunnel.
3.1. Initiating the Extended CONNECT Handshake

Endpoints using this mechanism to establish byte stream or datagram tunnels over HTTP/2 streams follow the CONNECT handshake procedure defined in [RFC6455]. However, instead of supplying "websocket" for the :protocol pseudo-header field to indicate a WebSocket connection, they specify "bytestream" or "datagram" to indicate a byte stream or datagram connection, respectively.

The :scheme and :path psuedo-headers are required by [RFC6455]. The scheme of the target URI MUST be set to "https" for both byte stream and datagram tunnels. The path is used in the same manner as for the WebSocket protocol, and MAY be set to "/" (an empty path component) if not desired for use.

Implementations should note that the Origin, Sec-WebSocket-Version, Sec-WebSocket-Protocol, and Sec-WebSocket-Extensions header fields are not included in the CONNECT request and response header fields, since this handshake mechanism is not being used to negotiate a WebSocket connection.

If the response to the extended CONNECT request indicates success of the handshake, then all further data sent or received on the new HTTP/2 stream is considered byte stream or datagram data.

3.2. Responding to the Extended CONNECT Handshake

A recipient of the extended CONNECT method follows the same procedure outlined by [RFC8441].

If the recipient encounters a :protocol pseudo-header with an unknown value or a value corresponding to a protocol they do not support, or if the recipient encounters violations of the extended CONNECT handshake protocol, they MUST return an HTTP response with an appropriate error code, such as 400 Bad Request. Otherwise, unknown header fields are ignored.

Once the handshake has been validated and is considered successful, the responder sends a HTTP response with status 200. After that response, all further data sent or received on the new HTTP/2 stream is considered byte stream or datagram data.

4. Using Tunnels Established via the Extended CONNECT Handshake

DATA frames are used as usual on the stream established by the CONNECT handshake to transmit data.
If the application negotiated the "bytestream" protocol, then individual DATA frames represent segments of an in-order bytestream and are delivered to the application as a stream of bytes. Implementations can deliver data to the application as soon as it becomes available, since there are no message boundaries to preserve.

If the application negotiated the "datagram" protocol, individual DATA frames are considered complete messages on the stream. Implementations SHOULD preserve these message boundaries when delivering data to the application. This prevents applications from needing to insert another level of framing to delineate message boundaries while transmitting datagram messages over HTTP/2 streams. Additionally, if an application is forwarding messages received over a "datagram" stream, the contents of each DATA frame should be sent in individual datagrams where possible.

The same considerations around intermediaries as defined in Section 7 of [RFC6455] apply to the extended CONNECT method, a client that connects via HTTP/2 to an HTTP proxy should use a traditional CONNECT request to tunnel through that proxy to the destination server. It should then

Streams created via the extended CONNECT method participate in flow control, stream prioritization, and other HTTP/2 features in the same manner as request and response streams defined in [RFC7540]. Stream closure continues to be interpreted as defined in Section 5 of [RFC8441].

Note that the frame type restrictions defined in Section 8.3 of [RFC7540] remain in effect: only DATA, RST_STREAM, WINDOW_UPDATE, and PRIORITY frames are allowed on the connected streams and any other frame types MUST be treated as a stream error (Section 5.4.2 of [RFC7540]) if received.

4.1. Example

An example of negotiating a "bytestream" stream on an HTTP/2 connection follows. This example is intended to closely follow the example in Section 5.1 of [RFC8441] to help illustrate the minor differences defined in this document.
5. IANA Considerations

This specification registers two entries in the "HTTP Upgrade Tokens" registry that was established by [RFC7230].

A new token, "bytestream", for byte stream data.

- Value: bytestream
- Description: Arbitrary bidirectional byte stream data
- Expected Version Tokens:
- References: [[RFC Editor: Please fill in this value with the RFC number for this document.]]

A new token, "datagram" for datagram data.

- Value: datagram
6. Security Considerations

The tunnels established by the CONNECT handshake are expected to be protected with a TLS connection. They inherit the security properties of this cryptographic context.

The security considerations of [RFC8441] Section 8 and [RFC7540] Section 10, and Section 10.5.2 especially, still apply to this use of the CONNECT method.

7. Acknowledgments

Thanks to Anthony Chivetta, Joshua Otto, and Valentin Pistol for their contributions in the design and implementation of this work.

8. Normative References


[RFC8441] McManus, P., "Bootstrapping WebSockets with HTTP/2",
RFC 8441, DOI 10.17487/RFC8441, September 2018,

Authors’ Addresses

Eric Kinnear
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: ekinnear@apple.com

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com
Security Considerations Regarding Compression Dictionaries
draft-kucherawy-httpbis-dict-sec-00

Abstract

Data compression algorithms benefit from blocks of tuning data called "dictionaries". These can greatly improve data compression speed and/or ratios, but their use and application has numerous potential security issues of concern to the communities using them. This document enumerates security issues known about compression dictionaries at the time of publication so that future proposals for use of dictionaries can benefit from this collected material.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 9, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction .................................................. 3
2. Definitions ................................................... 3
3. Dictionary Security Concerns ................................. 3
4. IANA Considerations ........................................... 4
5. Security Considerations ....................................... 4
6. References ..................................................... 5
   6.1. Normative References ..................................... 5
   6.2. Informative References ................................... 5
Appendix A. Acknowledgements ................................. 5
Appendix B. Prior Art ............................................. 5
1. Introduction

Brotli [RFC7932] and Zstandard [RFC8478] are examples of two modern data compression algorithms. While useful in their basic forms, they can be made far more effective with specific types of payloads when used with an object called a "dictionary". A dictionary is a map that can be applied during compression or uncompression that provides an advantage when operating against specific types of content. One might, for example, develop a dictionary that makes the compression algorithm more effective when applied to specific types of audio data.

As dictionaries are being developed, some issues have come to light that indicate ways that use of dictionaries might introduce destructive side effects to the environment in which their use is applied. This document is a collection of those topics, which can be consulted as work on dictionaries progresses; later, as RFCs are published advancing dictionaries, the content of this document could be used as a checklist to ensure that either the algorithms or their specification documents have been appropriately evaluated against these concerns.

2. 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.

3. Dictionary Security Concerns

These subsections each describe an issue that has been raised with respect to use of dictionaries as input to compression and uncompression. Where possible and known, acceptable mitigations are described.

[TODO: This is a bullet list for now, but each bullet item will gradually be converted into a subsection containing relevant discussion.]

- Attacks that use dictionary-based compression to recover content in the response.
- Attacks that use dictionary-based compression to recover content in the dictionary.
o Attacks that leverage dictionary-based compression to violate CORS/SOP/CSP. [need references and expansions for these]

o Attacks that manipulate a response’s content by manipulating the contents of a dictionary.

o Attacks that obfuscate a malicious response’s content through the use of dictionary-based compression.

o Attacks that identify users by fingerprinting their advertisement or use of dictionaries.

o Attacks that reveal past user behavior or associations through the negotiation and use of dictionaries.

o Attacks that use dictionaries to achieve denial-of-service / resource exhaustion:
  * against network resources
  * against storage resources
  * against computation resources
  * against the client
  * against the server
  * against an intermediary
  * against a third-party

o Inadvertent leakage of private information in the creation of dictionaries.

o General security risks that follow from complexity of implementation.

4. IANA Considerations

This document includes no actions for IANA.

[RFC Editor: Please remove this section before publication.]

5. Security Considerations

This document enumerates known security considerations about a space that is under development. The list of issues discussed above may
6. References

6.1. Normative References


6.2. Informative References


Appendix A. Acknowledgements

The author wishes to acknowledge the following for their review and constructive criticism of this update: TBD

Appendix B. Prior Art

Some prior art worth considering:

- draft-lee-sdch-spec, which was implemented in Chrome but then withdrawn
- draft-vkrasnov-h2-compression-dictionaries
- draft-vandevenne-shared-brotli-format
- HTTPBIS discussion during IETF 97
- Brotli "fetch spec" proposal: https://fetch.spec.whatwg.org/
o various HTTPBIS mailing list threads about dictionaries

Author’s Address

Murray S. Kucherawy
Facebook, Inc.
1 Hacker Way
Menlo Park, CA  94025
US

EMail: msk@fb.com
The Proxy-Status HTTP Header Field
draft-nottingham-proxy-status-00

Abstract

This document defines the Proxy-Status HTTP header field to convey the details of errors generated by HTTP intermediaries.

Note to Readers

RFC EDITOR: please remove this section before publication_

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/proxy-status [1].

The most recent (often, unpublished) draft is at https://mnot.github.io/I-D/proxy-status/ [2].

See also the draft’s current status in the IETF datatracker, at https://datatracker.ietf.org/doc/draft-nottingham-proxy-status/ [3].

Precursors that informed this work include (but are not limited to):

- https://docs.fastly.com/guides/debugging/common-503-errors
- https://docs.google.com/document/d/1fMEK80K1pHcL4CwhniOupgQu4MDsxK4y4dkthjjcMA

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 24, 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 Proxy-Status HTTP Header Field .......................... 4
   2.1. Generic Proxy Status Parameters ..................... 5
3. Proxy Status Types ...................................... 6
   3.1. DNS Timeout ........................................ 6
   3.2. DNS Error .......................................... 6
   3.3. Destination Not Found ............................... 6
   3.4. Destination Unavailable ............................. 7
   3.5. Destination IP Prohibited ........................... 7
   3.6. Destination IP Unroutable ............................ 7
   3.7. Connection Refused .................................. 7
   3.8. Connection Terminated ............................... 8
   3.9. Connection Timeout ................................ 8
   3.10. Connection Read Timeout ............................ 8
   3.11. Connection Write Timeout ........................... 8
   3.12. Connection Limit Reached ........................... 9
   3.13. HTTP Response Status .............................. 9
   3.14. HTTP Incomplete Response .......................... 9
   3.15. HTTP Protocol Error ................................ 9
   3.16. HTTP Response Header Block Too Large ............. 10
1. Introduction

HTTP intermediaries - including both forward proxies and gateways (also known as "reverse proxies") - have become an increasingly significant part of HTTP deployments. In particular, reverse proxies and Content Delivery Networks (CDNs) form part of the critical infrastructure of many Web sites.

Typically, HTTP intermediaries forward requests towards the origin server and then forward their responses back to clients. However, if an error occurs, the response is generated by the intermediary itself.

HTTP accommodates these types of errors with a few status codes; for example, 502 Bad Gateway and 504 Gateway Timeout. However, experience has shown that more information is necessary to aid debugging and communicate what’s happened to the client.

To address this, Section 2 defines a new HTTP response header field to convey such information, using the Proxy Status Types defined in Section 3. Section 4 explains how to define new Proxy Status Types.
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 Structured Headers [I-D.ietf-httpbis-header-structure] to specify syntax. The terms sh-param-list, sh-item, sh-string, sh-token and sh-integer refer to the structured types defined therein.

Note that in this specification, "proxy" is used to indicate both forward and reverse proxies, otherwise known as gateways. "Next hop" indicates the connection in the direction leading to the origin server for the request.

2. The Proxy-Status HTTP Header Field

The Proxy-Status HTTP response header field allows an intermediary to indicate the nature and details of an error condition it encounters when servicing a request.

It is a Structured Headers [I-D.ietf-httpbis-header-structure] Parameterised List, where each item in the list indicates an error condition. Typically, it will have only one param-item (the error condition that triggered generation of the response it occurs within), but more than one value is not prohibited.

Each param-item’s primary-id is a Proxy Status Type, a registered value that indicates the nature of the error.

Each param-item can have zero to many parameters. Section 2.1 lists parameters that can be used with all Proxy Status Types; individual types can define additional parameters to use with them. All parameters are optional; see Section 6 for their potential security impact.

For example:

HTTP/1.1 504 Gateway Timeout
Proxy-Status: connection_timeout; proxy=SomeCDN; origin=abc; tries=3

indicates the specific nature of the timeout as a connect timeout to the origin with the identifier "abc", and that it was generated by the intermediary that identifies itself as "FooCDN." Furthermore, three connection attempts were made.
Or:

HTTP/1.1 429 Too Many Requests
Proxy-Status: http_request_error; proxy=SomeReverseProxy

indicates that this 429 Too Many Requests response was generated by the intermediary, not the origin.

Each Proxy Status Type has a Recommended HTTP Status Code. When generating a HTTP response containing Proxy-Status, its HTTP status code SHOULD be set to the Recommended HTTP Status Code. However, there may be circumstances (e.g., for backwards compatibility with previous behaviours) when another status code might be used.

Section 3 lists the Proxy Status Types defined in this document; new ones can be defined using the procedure outlined in Section 4.

Proxy-Status MAY be sent in HTTP trailers, but - as with all trailers - it might be silently discarded along the path to the user agent, this SHOULD NOT be done unless it is not possible to send it in headers. For example, if an intermediary is streaming a response and the upstream connection suddenly terminates, Proxy-Status can be appended to the trailers of the outgoing message (since the headers have already been sent).

Note that there are various security considerations for intermediaries using the Proxy-Status header field; see Section 6.

Origin servers MUST NOT generate the Proxy-Status header field.

2.1. Generic Proxy Status Parameters

This section lists parameters that are potentially applicable to most Proxy Status Types.

- proxy - a sh-token identifying the HTTP intermediary generating this response.
- origin - a sh-token identifying the origin server whose behaviour triggered this response.
- protocol - a sh-token indicating the ALPN protocol identifier [RFC7301] used to connect to the next hop. This is only applicable when that connection was actually established.
- tries - a sh-integer indicating the number of times that the error has occurred before this response.
3. Proxy Status Types

This section lists the Proxy Status Types defined by this document. See Section 4 for information about defining new Proxy Status Types.

3.1. DNS Timeout

- **Name:** dns_timeout
- **Description:** The intermediary encountered a timeout when trying to find an IP address for the destination hostname.
- **Extra Parameters:** None.
- **Recommended HTTP status code:** 504

3.2. DNS Error

- **Name:** dns_error
- **Description:** The intermediary encountered a DNS error when trying to find an IP address for the destination hostname.
- **Extra Parameters:**
  - `rcode`: A `sh-string` conveying the DNS RCODE that indicates the error type. See [RFC8499], Section 3.
- **Recommended HTTP status code:** 502

3.3. Destination Not Found

- **Name:** destination_not_found
- **Description:** The intermediary cannot determine the appropriate destination to use for this request; for example, it may not be configured. Note that this error is specific to gateways, which typically require specific configuration to identify the "backend" server; forward proxies use in-band information to identify the origin server.
- **Extra Parameters:** None.
- **Recommended HTTP status code:** 500
3.4. Destination Unavailable
   o Name: destination_unavailable
   o Description: The intermediary considers the next hop to be unavailable; e.g., recent attempts to communicate with it may have failed, or a health check may indicate that it is down.
   o Extra Parameters:
   o Recommended HTTP status code: 503

3.5. Destination IP Prohibited
   o Name: destination_ip_prohibited
   o Description: The intermediary is configured to prohibit connections to the destination IP address.
   o Extra Parameters: None.
   o Recommended HTTP status code: 502

3.6. Destination IP Unroutable
   o Name: destination_ip_unroutable
   o Description: The intermediary cannot find a route to the destination IP address.
   o Extra Parameters: None.
   o Recommended HTTP status code: 502

3.7. Connection Refused
   o Name: connection_refused
   o Description: The intermediary’s connection to the next hop was refused.
   o Extra Parameters: None.
   o Recommended HTTP status code: 502
3.8. Connection Terminated
   o Name: connection_terminated
   o Description: The intermediary’s connection to the next hop was closed before any part of the response was received. If some part was received, see http_response_incomplete.
   o Extra Parameters: None.
   o Recommended HTTP status code: 502

3.9. Connection Timeout
   o Name: connection_timeout
   o Description: The intermediary’s attempt to open a connection to the next hop timed out.
   o Extra Parameters: None.
   o Recommended HTTP status code: 504

3.10. Connection Read Timeout
   o Name: connection_read_timeout
   o Description: The intermediary was expecting data on a connection (e.g., part of a response), but did not receive any new data in a configured time limit.
   o Extra Parameters: None.
   o Recommended HTTP status code: 504

3.11. Connection Write Timeout
   o Name: connection_write_timeout
   o Description: The intermediary was attempting to write data to a connection, but was not able to (e.g., because its buffers were full).
   o Extra Parameters: None.
   o Recommended HTTP status code: 504
3.12. Connection Limit Reached

- Name: connection_limit_reached
- Description: The intermediary is configured to limit the number of connections it has to the next hop, and that limit has been passed.
- Extra Parameters: None.
- Recommended HTTP status code:

3.13. HTTP Response Status

- Name: http_response_status
- Description: The intermediary has received a 4xx or 5xx status code from the next hop and forwarded it to the client.
- Extra Parameters: None.
- Recommended HTTP status code:

3.14. HTTP Incomplete Response

- Name: http_response_incomplete
- Description: The intermediary received an incomplete response to the request from the next hop.
- Extra Parameters: None.
- Recommended HTTP status code: 502

3.15. HTTP Protocol Error

- Name: http_protocol_error
- Description: The intermediary encountered a HTTP protocol error when communicating with the next hop. This error should only be used when a more specific one is not defined.
- Extra Parameters:
  * details: a sh-string containing details about the error condition. For example, this might be the HTTP/2 error code or free-form text describing the condition.
Recommended HTTP status code: 502

3.16. HTTP Response Header Block Too Large

- Name: http_response_header_block_size
- Description: The intermediary received a response to the request whose header block was considered too large.
- Extra Parameters:
  * header_block_size: a sh-integer indicating how large the headers received were. Note that they might not be complete; i.e., the intermediary may have discarded or refused additional data.

Recommended HTTP status code: 502

3.17. HTTP Response Header Too Large

- Name: http_response_header_size
- Description: The intermediary received a response to the request containing an individual header line that was considered too large.
- Extra Parameters:
  * header_name: a sh-string indicating the name of the header that triggered the error.

Recommended HTTP status code: 502

3.18. HTTP Response Body Too Large

- Name: http_response_body_size
- Description: The intermediary received a response to the request whose body was considered too large.
- Extra Parameters:
  * body_size: a sh-integer indicating how large the body received was. Note that it may not have been complete; i.e., the intermediary may have discarded or refused additional data.

Recommended HTTP status code: 502
3.19. HTTP Response Transfer-Coding Error
   - Name: http_response_transfer_coding
   - Description: The intermediary encountered an error decoding the transfer-coding of the response.
   - Extra Parameters:
     * coding: a sh-token containing the specific coding that caused the error.
     * details: a sh-string containing details about the error condition.
   - Recommended HTTP status code: 502

3.20. HTTP Response Content-Coding Error
   - Name: http_response_content_coding
   - Description: The intermediary encountered an error decoding the content-coding of the response.
   - Extra Parameters:
     * coding: a sh-token containing the specific coding that caused the error.
     * details: a sh-string containing details about the error condition.
   - Recommended HTTP status code: 502

3.21. HTTP Response Timeout
   - Name: http_response_timeout
   - Description: The intermediary reached a configured time limit waiting for the complete response.
   - Extra Parameters: None.
   - Recommended HTTP status code: 504
3.22. TLS Handshake Error

- Name: tls_handshake_error
- Description: The intermediary encountered an error during TLS handshake with the next hop.
- Extra Parameters:
  * alert_message: a sh-token containing the applicable description string from the TLS Alerts registry.
- Recommended HTTP status code: 502

3.23. TLS Untrusted Peer Certificate

- Name: tls_untrusted_peer_certificate
- Description: The intermediary received untrusted peer certificate during TLS handshake with the next hop.
- Extra Parameters: None.
- Recommended HTTP status code: 502

3.24. TLS Expired Peer Certificate

- Name: tls_expired_peer_certificate
- Description: The intermediary received expired peer certificate during TLS handshake with the next hop.
- Extra Parameters: None.
- Recommended HTTP status code: 502

3.25. TLS Unexpected Peer Certificate

- Name: tls_unexpected_peer_certificate
- Description: The intermediary received unexpected peer certificate (e.g., SPKI doesn’t match) during TLS handshake with the next hop.
- Extra Parameters:
  * details: a sh-string containing the checksum or SPKI of the certificate received from the next hop.
3.26. TLS Unexpected Peer Identity

- Name: tls_unexpected_peer_identity

- Description: The intermediary received peer certificate with unexpected identity (e.g., Subject Alternative Name doesn’t match) during TLS handshake with the next hop.

- Extra Parameters:
  
  * details: a sh-string containing the identity of the next hop.

- Recommended HTTP status code: 502

3.27. TLS Missing Proxy Certificate

- Name: tls_missing_proxy_certificate

- Description: The next hop requested client certificate from the intermediary during TLS handshake, but it wasn’t configured with one.

- Extra Parameters: None.

- Recommended HTTP status code: 500

3.28. TLS Rejected Proxy Certificate

- Name: tls_rejected_proxy_certificate

- Description: The next hop rejected client certificate provided by the intermediary during TLS handshake.

- Extra Parameters: None.

- Recommended HTTP status code: 500

3.29. TLS Error

- Name: tls_error

- Description: The intermediary encountered a TLS error when communicating with the next hop.

- Extra Parameters:
* alert_message: a sh-token containing the applicable description string from the TLS Alerts registry.

  o Recommended HTTP status code: 502

3.30. HTTP Request Error

  o Name: http_request_error

  o Description: The intermediary is generating a client (4xx) response on the origin’s behalf. Applicable status codes include (but are not limited to) 400, 403, 405, 406, 408, 411, 413, 414, 415, 416, 417, 429. This proxy status type helps distinguish between responses generated by intermediaries from those generated by the origin.

  o Extra Parameters: None.

  o Recommended HTTP status code: The applicable 4xx status code

3.31. HTTP Request Denied

  o Name: http_request_denied

  o Description: The intermediary rejected HTTP request based on its configuration and/or policy settings. The request wasn’t forwarded to the next hop.

  o Extra Parameters: None.

  o Recommended HTTP status code: 400

3.32. HTTP Upgrade Failed

  o Name: http_upgrade_failed

  o Description: The HTTP Upgrade between the intermediary and the next hop failed.

  o Extra Parameters: None.

  o Recommended HTTP status code: 502

3.33. Proxy Internal Error

  o Name: proxy_internal_error
o Description: The intermediary encountered an internal error unrelated to the origin.

o Extra Parameters:
  * details: a sh-string containing details about the error condition.

o Recommended HTTP status code: 500

4. Defining New Proxy Status Types

New Proxy Status Types can be defined by registering them in the HTTP Proxy Status Types registry.

Registration requests are reviewed and approved by a Designated Expert, as per [RFC8126] Section 4.5. A specification document is appreciated, but not required.

The Expert(s) should consider the following factors when evaluating requests:

o Community feedback

o If the value is sufficiently well-defined

o If the value is generic; vendor-specific, application-specific and deployment-specific values are discouraged

Registration requests should use the following template:

o Name: [a name for the Proxy Status Type that is allowable as a sh-param-list key]

o Description: [a description of the conditions that generate the Proxy Status Types]

o Extra Parameters: [zero or more optional parameters, typed using one of the types available in sh-item]

o Recommended HTTP status code: [the appropriate HTTP status code for this entry]

5. IANA Considerations

Upon publication, please create the HTTP Proxy Status Types registry at https://iana.org/assignments/http-proxy-statuses [5] and populate it with the types defined in Section 3; see Section 4 for its associated procedures.

6. Security Considerations

One of the primary security concerns when using Proxy-Status is leaking information that might aid an attacker.

As a result, care needs to be taken when deciding to generate a Proxy-Status header. Note that intermediaries are not required to generate a Proxy-Status header field in any response, and can conditionally generate them based upon request attributes (e.g., authentication tokens, IP address).

Likewise, generation of all parameters is optional.

Special care needs to be taken in generating proxy and origin parameters, as they can expose information about the intermediary’s configuration and back-end topology.

7. References

7.1. Normative References

[I-D.ietf-httpbis-header-structure]


7.2. URIs


Authors' Addresses

Mark Nottingham
Fastly

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

Piotr Sikora
Google

Email: piotrsikora@google.com
Abstract

Content providers delivering content via CDNs will sometimes deliver content over HTTPS (or both HTTPS and HTTP) but configure the CDN to pull from the origin over cleartext and unauthenticated HTTP. From the perspective of a client, it appears that their requests and associated responses are delivered over HTTPS, while in reality their requests are being sent across the network in-the-clear and responses are delivered unauthenticated. This exposes user request data to pervasive monitoring [RFC7258]; it also means response data may be tampered with by active adversaries. Terminating TLS connections on a load balancer and contacting a backend over cleartext has long been common within data centers, but doing this TLS termination and downgrade to HTTP at a CDN introduces additional risk when the unprotected traffic is sent over the general Internet, sometimes across national boundaries.

While it would be nice to say "never do this," customer demand, content provider use-cases, and market forces today make it impossible for CDNs to not support downgrade. However, following a set of best practices can provide visibility into when this is happening and can reduce some of the risks.

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."
1. Background and Motivation

Browsers are helping drive a push to universal HTTPS through a variety of mechanisms, including:

- Show HTTP as "not secure"
- Showing mixed-content warnings when images or advertisements are HTTP on an HTTPS base page
- Making "powerful" new web features available only for HTTPS

On mobile, app stores sometimes require HTTPS for acceptance.

These factors have pushed many content providers to quickly enable HTTPS, even when their origin infrastructure is not ready or not
perceived as being ready. Being able to use a CDN to convert HTTPS to HTTP has been looked at as a fast path for getting onto HTTPS quickly. Doing this has value in protecting requests and responses over the last mile, but admittedly does not address or worsens some other types of attacks (such as pervasive monitoring, or corruption and manipulation of content crossing national boundaries).

Delivering content over HTTPS but fetching it insecurely over HTTP is done for a variety of reasons, some of which have historic motivations with better alternatives today, but where content providers are resistant to change. This includes:

- Lack of HTTPS support in origin infrastructure, such as due to using load balancing hardware that does not support HTTPS, has bad performance characteristics with HTTPS, or which only supports SSLv3.

- A perception that HTTPS is more expensive to deliver. In some cases content providers may have origin infrastructure using old hardware where this is a real challenge and they lack the budget to upgrade to servers or load balancers that can handle HTTPS well.

- A perception that using HTTPS introduces performance issues, such as due to the additional round trips required to establish connections. This can be a real issue for origins that lack persistent connection or session resumption support.

- Challenges in managing origin certificates, or a perception that it is hard to get TLS certificates. Automation with providers such as LetsEncrypt help here, but some content provider origins may be using software or hardware elements that don’t yet integrate well with Auto-DV or may have organizational policies against using DV certificates.

- Delivering the same library of content to end-users over both HTTP and HTTPS, but wanting the CDN to cache any given object only once. This can be better addressed by always fetching content via HTTPS and storing in a cache accessible for both HTTP and HTTPS requests, but this faces challenges for transitioning from an entirely HTTP-fetched-and-served content library to one that is served over a mixture of HTTP and HTTPS.

- A perception that there is no risk to their users or brand reputation, sometimes due to thinking of pervasive monitoring and content manipulation as esoteric threats that don’t apply in their case. For example, content providers delivering on-demand
streaming movies may not see a threat from using HTTP and may view DRM as addressing most of their immediate concerns.

There is also a closely-related issue where content delivered over HTTPS has been pushed to origin infrastructure over an insecure protocol. For example, content uploaded to a storage service over an insecure protocol such as FTP, or live streams pushed from encoders to ingest entry points over an insecure protocol. This has the added risk that authenticators may be unprotected on-the-wire.

2. Recommended alternatives

The "right thing" to do is to use modern secure protocols and cryptography for secrecy and authentication for the request and the response when interacting with content origin sources: HTTPS for pull-through caches, and protocols such as SCP or SFTP or FTP-over-TLS or HTTPS POSTs for pushed data.

Origin sites that avoided TLS for fear of a performance hit should collect data on the actual costs with modern implementations and modern crypto-support hardware. These are expected to be under 2% CPU overhead, especially when persistent connections are enabled. Auto-DV certificate management can make origin certificate management straight-forward and automateable.

3. Potential risk mitigations

An intermediate cache can take several actions to reduce the risk of unpleasant consequences from using TLS downgrade - though these practices do not eliminate that risk. They take two general strategies:

1. Informing the endpoints that this downgrade is in place. End points have more information about the details of the connection, and can expose details to human controllers. For example, returning a response header such as "Protocol-To-Origin: cleartext" and preventing customers from removing it. Clients may then choose some manner in which to expose this to end-users. (Some other proprietary implementations of this response header have included "X-Forward-Proto: http" and "CDN-Origin-Protocol: http").

2. Restricting the sort of data in transit when downgrading from HTTPS to cleartext HTTP. Examples of this include:

   * Limiting to GET methods. This prevents unauthenticated writes to the origin.
* Refusing to downgrade requests for "/", "/index/", or "/index.html". This prevents accidental delivery of the entire site. The goal is to rapidly detect a misconfiguration with too much downgrading by breaking the site.

* Limiting the content types or file extensions (e.g., to streaming media or other static media assets).

* Stripping outgoing request headers containing potential identifiers (Cookie, etc)

* Stripping query strings

In practice, stripping query strings breaks an enormous amount of Web traffic: searches, beacons, and the selection apparatus of streaming media clients. Mechanisms that rely on lists of what is allowed (file extensions) or what is banned (such as "Cookie" headers) rely on an implausibly detailed and up-to-date models of Web use.

Other headers that may wish to be stripped from outgoing requests include "X-Forwarded-For", "Origin", "Referer", "Cookie", "Cookie2", and those starting with "Sec-" or "Proxy-".

4. Recommendations

It is recommended that CDNs do at least the following as default behaviors as part of TLS downgrade:

o Providing and encouraging better alternatives (such as always fetching securely over HTTPS but making static objects available in a shared cache that can also be accessed via HTTP requests).

o Returning a "Protocol-To-Origin: cleartext" response header (which may be a comma-separated list of protocols when multiple hops are involved).

o Limiting downgrade requests to GET.

o Refusing requests for "/", "/index/", or "/index.html".

o Strip at least some headers that may include personal identifiers or sensitive information.

5. Alternative approaches

Some other approaches may also help address the risks:
o Use a VPN or IPSEC or other secure channel between the CDN and the origin.

o Validate asymmetric signatures of content at the CDN before serving, such as for software downloads. This helps with integrity, but still exposes confidentiality risks.

6. Security Considerations

6.1. Risks of doing downgrades

Downgrades allow protection of last-mile connections to end-users, but they make it easier for adversaries who control the network between CDN caches and origin (such as at national boundaries) to poison caches or perform surveillance (as correlation attacks are possible, even if ostensible PII information is stripped at the CDN.)

6.2. Control of the network between the cache and the origin

ISPs on the HTTP path, including nation states at their borders, can surveil traffic. They can expect to get end-user IP information from "X-Forwarded-For" or similar. In some circumstances, they can learn more from correlated timing and sizes. This is principally a risk to _secrecy_.

Active adversaries can also corrupt or modify content.

For executable content (such as software downloads or javascript) this can be used to compromise clients or web pages, especially if no end-to-end secure integrity validation is performed. Even when software downloads have signature validation performed, this can provide a potential exposure for downgrade attacks, depending on client-side implementations.

For site and media content, modification can be used to make content appear as authoritative to a user (delivered via HTTPS from a "trusted site") while actually containing selective modifications of the attackers choice, such as for the financial or political benefit of the attacker.

6.3. Confused-deputy issues at the browser or origin

HTTP clients make different decisions based on whether they are using HTTPS or HTTP - for example, they send Secure cookies (cite), they enable certain Web features (high-resolution location, Service Workers). This is principally a risk to _authentication_.

Sniffen, et al. Expires September 25, 2019
This attack is only available with downgrade. A related attack is available in the case of HTTP _upgrade_, in which a server makes a similar decision based on seeing HTTPS on its end of the connection. In cases where HTTP requests are upgraded to HTTPS, CDN or proxy operators need to work with origin operators to control this complexity and prevent the complementary attack, such as by only performing upgrades for cache-able, static, and idempotent content.

7. Normative References


Appendix A. Acknowledgements

Thank you to Suneeth Jayan and others at Akamai who have helped develop best practices. Future versions of this draft hope to also incorporate best practices developed elsewhere.

Authors’ Addresses

Brian Sniffen
Akamai Technologies
145 Broadway
Cambridge  02139
US

Email: bsniffen@akamai.com

Mike Bishop
Akamai Technologies
145 Broadway
Cambridge  02139
US

Email: mbishop@akamai.com

Erik Nygren
Akamai Technologies
145 Broadway
Cambridge  02139
US

Email: nygren@akamai.com
Signed HTTP Exchanges
draft-yasskin-http-origin-signed-responses-05

Abstract

This document specifies how a server can send an HTTP exchange—a request URL, content negotiation information, and a response—with signatures that vouch for that exchange’s authenticity. These signatures can be verified against an origin’s certificate to establish that the exchange is authoritative for an origin even if it was transferred over a connection that isn’t. The signatures can also be used in other ways described in the appendices.

These signatures contain countermeasures against downgrade and protocol-confusion attacks.

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].

The source code and issues list for this draft can be found in https://github.com/WICG/webpackage [2].

Status of This Memo

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

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

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

This Internet-Draft will expire on July 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 ........................................... 4
2. Terminology ........................................... 4
3. Signing an exchange .................................. 5
   3.1. The Signature Header ............................ 6
       3.1.1. Examples .................................. 7
       3.1.2. Open Questions ............................ 8
   3.2. CBOR representation of exchange response headers ... 9
       3.2.1. Example .................................. 9
   3.3. Loading a certificate chain ........................ 10
   3.4. Canonical CBOR serialization ..................... 11
   3.5. Signature validity ............................... 11
       3.5.1. Open Questions ............................ 16
   3.6. Updating signature validity ....................... 16
       3.6.1. Examples .................................. 17
   3.7. The Accept-Signature header ........................ 18
       3.7.1. Integrity identifiers ..................... 19
       3.7.2. Key type identifiers ..................... 19
       3.7.3. Key value identifiers ..................... 20
       3.7.4. Examples .................................. 20
       3.7.5. Open Questions ............................ 21
4. Cross-origin trust ..................................... 21
   4.1. Uncached header fields ............................ 23
       4.1.1. Stateful header fields ..................... 23
   4.2. Certificate Requirements .......................... 24
5. Transferring a signed exchange ....................... 25
   5.1. Same-origin response ............................. 25
       5.1.1. Serialized headers for a same-origin response ... 26
       5.1.2. The Signed-Headers Header .................. 26
   5.2. HTTP/2 extension for cross-origin Server Push ...... 27
       5.2.1. Indicating support for cross-origin Server Push ... 27
       5.2.2. NO_TRUSTED_EXCHANGE_SIGNATURE error code ....... 27
5.2.3. Validating a cross-origin Push .............................. 28
5.3. application/signed-exchange format ............................ 29
  5.3.1. Cross-origin trust in application/signed-exchange .......... 30
  5.3.2. Example .................................................. 30
  5.3.3. Open Questions ............................................ 30
6. Security considerations ........................................... 31
  6.1. Over-signing .................................................. 31
    6.1.1. Session fixation ......................................... 31
    6.1.2. Misleading content ...................................... 31
  6.2. Off-path attackers ........................................... 32
  6.3. Downgrades ................................................... 32
  6.4. Signing oracles are permanent .................................. 32
  6.5. Unsigned headers ............................................ 32
  6.6. application/signed-exchange ................................... 33
  6.7. Key re-use with TLS .......................................... 33
  6.8. Content sniffing .............................................. 34
7. Privacy considerations ............................................ 35
8. IANA considerations .............................................. 35
  8.1. Signature Header Field Registration .......................... 35
  8.2. Accept-Signature Header Field Registration ..................... 36
  8.3. Signed-Headers Header Field Registration ....................... 36
  8.4. HTTP/2 Settings .............................................. 36
  8.5. HTTP/2 Error code ............................................ 37
  8.6. Internet Media Type application/signed-exchange ................ 37
  8.7. Internet Media Type application/cert-chain+cbor ................ 38
9. References .......................................................... 39
  9.1. Normative References .......................................... 39
  9.2. Informative References ........................................ 41
  9.3. URIs .......................................................... 44
Appendix A. Use cases ............................................... 44
  A.1. PUSHed subresources .......................................... 44
  A.2. Explicit use of a content distributor for subresources ......... 45
  A.3. Subresource Integrity ......................................... 46
  A.4. Binary Transparency .......................................... 46
  A.5. Static Analysis ............................................... 46
  A.6. Offline websites .............................................. 47
Appendix B. Requirements ............................................. 47
  B.1. Proof of origin ............................................... 47
    B.1.1. Certificate constraints .................................... 47
    B.1.2. Signature constraints .................................... 47
    B.1.3. Retrieving the certificate ................................ 48
    B.2. How much to sign ............................................ 48
      B.2.1. Conveying the signed headers ........................... 49
    B.3. Response lifespan ............................................ 49
      B.3.1. Certificate revocation .................................... 50
      B.3.2. Response downgrade attacks ................................ 50
    B.4. Low implementation complexity ................................ 51
      B.4.1. Limited choices ........................................ 51
1. Introduction

Signed HTTP exchanges provide a way to prove the authenticity of a resource in cases where the transport layer isn’t sufficient. This can be used in several ways:

- When signed by a certificate ([RFC5280]) that’s trusted for an origin, an exchange can be treated as authoritative for that origin, even if it was transferred over a connection that isn’t authoritative (Section 9.1 of [RFC7230]) for that origin. See Appendix A.1 and Appendix A.2.

- A top-level resource can use a public key to identify an expected publisher for particular subresources, a system known as Subresource Integrity ([SRI]). An exchange’s signature provides the matching proof of authorship. See Appendix A.3.

- A signature can vouch for the exchange in some way, for example that it appears in a transparency log or that static analysis indicates that it omits certain attacks. See Appendix A.4 and Appendix A.5.

Subsequent work toward the use cases in [I-D.yasskin-webpackage-use-cases] will provide a way to group signed exchanges into bundles that can be transmitted and stored together, but single signed exchanges are useful enough to standardize on their own.

2. Terminology

Absolute URL A string for which the URL parser [3] ([URL]), when run without a base URL, returns a URL rather than a failure, and for which that URL has a null fragment. This is similar to the absolute-URL string [4] concept defined by ([URL]) but might not include exactly the same strings.

Author The entity that wrote the content in a particular resource. This specification deals with publishers rather than authors.
Publisher  The entity that controls the server for a particular origin [RFC6454]. The publisher can get a CA to issue certificates for their private keys and can run a TLS server for their origin.

Exchange (noun)  An HTTP request URL, content negotiation information, and an HTTP response. This can be encoded into a request message from a client with its matching response from a server, into the request in a PUSH_PROMISE with its matching response stream, or into the dedicated format in Section 5.3, which uses [I-D.ietf-httpbis-variants] to encode the content negotiation information. This is not quite the same meaning as defined by Section 8 of [RFC7540], which assumes the content negotiation information is embedded into HTTP request headers.

Intermediate  An entity that fetches signed HTTP exchanges from a publisher or another intermediate and forwards them to another intermediate or a client.

Client  An entity that uses a signed HTTP exchange and needs to be able to prove that the publisher vouched for it as coming from its claimed origin.

Unix time  Defined by [POSIX] section 4.16 [5].

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. Signing an exchange

In the response of an HTTP exchange the server MAY include a "Signature" header field (Section 3.1) holding a list of one or more parameterised signatures that vouch for the content of the exchange. Exactly which content the signature vouches for can depend on how the exchange is transferred (Section 5).

The client categorizes each signature as "valid" or "invalid" by validating that signature with its certificate or public key and other metadata against the exchange’s URL, response headers, and content (Section 3.5). This validity then informs higher-level protocols.

Each signature is parameterised with information to let a client fetch assurance that a signed exchange is still valid, in the face of revoked certificates and newly-discovered vulnerabilities. This
assurance can be bundled back into the signed exchange and forwarded to another client, which won’t have to re-fetch this validity information for some period of time.

3.1. The Signature Header

The "Signature" header field conveys a list of signatures for an exchange, each one accompanied by information about how to determine the authority of and refresh that signature. Each signature directly signs the exchange’s URL and response headers and identifies one of those headers that enforces the integrity of the exchange’s payload.

The "Signature" header is a Structured Header as defined by [I-D.ietf-httpbis-header-structure]. Its value MUST be a parameterised list (Section 3.4 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

```
Signature = sh-param-list
```

Each parameterised identifier in the list MUST have parameters named "sig", "integrity", "validity-url", "date", and "expires". Each parameterised identifier MUST also have either "cert-url" and "cert-sha256" parameters or an "ed25519key" parameter. This specification gives no meaning to the identifier itself, which can be used as a human-readable identifier for the signature (see Section 3.1.2, Paragraph 1). The present parameters MUST have the following values:

"sig" Byte sequence (Section 3.10 of [I-D.ietf-httpbis-header-structure]) holding the signature of most of these parameters and the exchange’s URL and response headers.

"integrity" A string (Section 3.8 of [I-D.ietf-httpbis-header-structure]) containing a "/"-separated sequence of names starting with the lowercase name of the response header field that guards the response payload’s integrity. The meaning of subsequent names depends on the response header field, but for the "digest" header field, the single following name is the name of the digest algorithm that guards the payload’s integrity.

"cert-url" A string (Section 3.8 of [I-D.ietf-httpbis-header-structure]) containing an absolute URL (Section 2) with a scheme of "https" or "data".

"cert-sha256" Byte sequence (Section 3.10 of [I-D.ietf-httpbis-header-structure]) holding the SHA-256 hash of the first certificate found at "cert-url".
"ed25519key" Byte sequence (Section 3.10 of [I-D.ietf-httpbis-header-structure]) holding an Ed25519 public key ([RFC8032]).

"validity-url" A string (Section 3.8 of [I-D.ietf-httpbis-header-structure]) containing an absolute URL (Section 2) with a scheme of "https".

"date" and "expires" An integer (Section 3.6 of [I-D.ietf-httpbis-header-structure]) representing a Unix time.

The "cert-url" parameter is _not_ signed, so intermediates can update it with a pointer to a cached version.

3.1.1. Examples

The following header is included in the response for an exchange with effective request URI "https://example.com/resource.html". Newlines are added for readability.

```
Signature:
sig1; sig=*MEUCIQXlI2qN3NB1gFiuRTNFpZxDEIAUpX6HIEwcZEc0cZLYA1ga9DsVOMM+g5XpwEBdGW3s+S+bvnmAUIJSWwhuBdq5UY=*
integrity="digest/mi-sha256";
validity-url="https://example.com/resource.validity.1511128380";
cert-url="https://example.com/oldcerts";
cert-sha256=*W7uB969dFW3Mb5ZeffPS9Tq5ZbH5iSmOIlpjjv2qEArmI=*
date=1511128380; expires=1511733180,
sig2;
sig=MEQCIGjZrqTRf9iKNkGFyzRMTFgwf/brY2ZNIP/dykhUV0aYAIbTXg+8wujoT4n/W+cNg7b7pGqQvIUGY28u8HZJ5YH26Qg=*
integrity="digest/mi-sha256";
validity-url="https://example.com/resource.validity.1511128380";
cert-url="https://example.com/newcerts";
cert-sha256=Ji1Em9KnR0DcMnINbVitpvdYKNQ+YgBj99D1y4fEXw=*
date=1511128380; expires=1511733180,
sr=sigs;
se=*
```

```
sig=*IGZVaJMM5f2oGqZC1mbDkTDL+ QUADza4BgeO494ggACYJOvrof6uh50JCCwKr7DK+LBch0js sDYpp5CLc1SDA=* 
integrity="digest/mi-sha256"; 
validity-url="https://example.com/resource.validity.1511128380"; 
ed25519key=*zsSevyFsxyZHiULvBd4eypdRLTyqWRVOJuukUZ+a8=* 
date=1511128380; expires=1511733180, 
thirdparty;
sig=MEYCIQCNwZjn6R2fNxsobktir8TkiaJYQFhWTuW11i4PewQaQhAMs2TVj4rTshDtx5bgQEO wggj2mRXLhFXZtXgPupi=*; 
integrity="digest/mi-sha256"; 
validity-url="https://thirdparty.example.com/resource.validity.1511161860"; 
cert-url="https://thirdparty.example.com/certs"; 
cert-sha256=UeOwUPkvxlRGTyvHcsmMUN0A2oNsZbU8EUv8A9ZAnNc=*
```
There are 4 signatures: 2 from different secp256r1 certificates within "https://example.com/", one using a raw ed25519 public key that’s also controlled by "example.com", and a fourth using a secp256r1 certificate owned by "thirdparty.example.com".

All 4 signatures rely on the "Digest" response header with the mi-sha256 digest algorithm to guard the integrity of the response payload.

The signatures include a "validity-url" that includes the first time the resource was seen. This allows multiple versions of a resource at the same URL to be updated with new signatures, which allows clients to avoid transferring extra data while the old versions don’t have known security bugs.

The certificates at "https://example.com/oldcerts" and "https://example.com/newcerts" have "subjectAltName"s of "example.com", meaning that if they and their signatures validate, the exchange can be trusted as having an origin of "https://example.com/". The publisher might be using two certificates because their readers have disjoint sets of roots in their trust stores.

The publisher signed with all three certificates at the same time, so they share a validity range: 7 days starting at 2017-11-19 21:53 UTC.

The publisher then requested an additional signature from "thirdparty.example.com", which did some validation or processing and then signed the resource at 2017-11-19 23:11 UTC. "thirdparty.example.com" only grants 4-day signatures, so clients will need to re-validate more often.

3.1.2. Open Questions

[I-D.ietf-httpbis-header-structure] provides a way to parameterise identifiers but not other supported types like byte sequences. If the "Signature" header field is notionally a list of parameterised signatures, maybe we should add a "parameterised byte sequence" type.

Should the cert-url and validity-url be lists so that intermediates can offer a cache without losing the original URLs? Putting lists in dictionary fields is more complex than [I-D.ietf-httpbis-header-structure] allows, so they’re single items for now.
3.2. CBOR representation of exchange response headers

To sign an exchange’s response headers, they need to be serialized into a byte string. Since intermediaries and distributors (Appendix A.2) might rearrange, add, or just reserialize headers, we can’t use the literal bytes of the headers as this serialization. Instead, this section defines a CBOR representation that can be embedded into other CBOR, canonically serialized (Section 3.4), and then signed.

The CBOR representation of a set of response metadata and headers is the CBOR ([RFC7049]) map with the following mappings:

- The byte string ':status' to the byte string containing the response’s 3-digit status code, and
- For each response header field, the header field’s lowercase name as a byte string to the header field’s value as a byte string.

3.2.1. Example

Given the HTTP exchange:

GET / HTTP/1.1
Host: example.com
Accept: */*

HTTP/1.1 200
Content-Type: text/html
Digest: mi-sha256=dcRDgR2GM35DluAV13PzgnG6+pvQwPywfFvAu1UeFrs=
Signed-Headers: "content-type", "digest"

<!doctype html>
<html>
...

The cbor representation consists of the following item, represented using the extended diagnostic notation from [I-D.ietf-cbor-cddl] appendix G:

```json
{
    'digest': 'mi-sha256=dcRDgR2GM35DluAV13PzgnG6+pvQwPywfFvAu1UeFrs=',
    ':status': '200',
    'content-type': 'text/html'
}
```
3.3. Loading a certificate chain

The resource at a signature’s "cert-url" MUST have the "application/cert-chain+cbor" content type, MUST be canonically-encoded CBOR (Section 3.4), and MUST match the following CDDL:


The first map (second item) in the CBOR array is treated as the end-entity certificate, and the client will attempt to build a path ([RFC5280]) to it from a trusted root using the other certificates in the chain.

1. Each "cert" value MUST be a DER-encoded X.509v3 certificate ([RFC5280]). Other key/value pairs in the same array item define properties of this certificate.

2. The first certificate’s "ocsp" value MUST be a complete, DER-encoded OCSP response for that certificate (using the ASN.1 type "OCSPResponse" defined in [RFC6960]). Subsequent certificates MUST NOT have an "ocsp" value.

3. Each certificate’s "sct" value if any MUST be a "SignedCertificateTimestampList" for that certificate as defined by Section 3.3 of [RFC6962].

Loading a "cert-url" takes a "forceFetch" flag. The client MUST:

1. Let "raw-chain" be the result of fetching ([FETCH]) "cert-url". If "forceFetch" is _not_ set, the fetch can be fulfilled from a cache using normal HTTP semantics [RFC7234]. If this fetch fails, return "invalid".

2. Let "certificate-chain" be the array of certificates and properties produced by parsing "raw-chain" using the CDDL above. If any of the requirements above aren’t satisfied, return "invalid". Note that this validation requirement might be impractical to completely achieve due to certificate validation implementations that don’t enforce DER encoding or other standard constraints.
3. Return "certificate-chain".

3.4. Canonical CBOR serialization

Within this specification, the canonical serialization of a CBOR item uses the following rules derived from Section 3.9 of [RFC7049] with erratum 4964 applied:

- Integers and the lengths of arrays, maps, and strings MUST use the smallest possible encoding.
- Items MUST NOT be encoded with indefinite length.
- The keys in every map MUST be sorted in the bytewise lexicographic order of their canonical encodings. For example, the following keys are correctly sorted:

  1. 10, encoded as 0A.
  2. 100, encoded as 18 64.
  3. -1, encoded as 20.
  4. "x", encoded as 61 7A.
  5. "aa", encoded as 62 61 61.
  6. [100], encoded as 81 18 64.
  7. [-1], encoded as 81 20.
  8. false, encoded as F4.

Note: this specification does not use floating point, tags, or other more complex data types, so it doesn’t need rules to canonicalize those.

3.5. Signature validity

The client MUST parse the "Signature" header field as the parameterised list (Section 4.2.5 of [I-D.ietf-httpbis-header-structure]) described in Section 3.1. If an error is thrown during this parsing or any of the requirements described there aren’t satisfied, the exchange has no valid signatures. Otherwise, each member of this list represents a signature with parameters.
The client MUST use the following algorithm to determine whether each signature with parameters is invalid or potentially-valid for an exchange’s

- "requestUrl", a byte sequence that can be parsed into the exchange’s effective request URI (Section 5.5 of [RFC7230]),
- "responseHeaders", a byte sequence holding the canonical serialization (Section 3.4) of the CBOR representation (Section 3.2) of the exchange’s response metadata and headers, and
- "payload", a stream of bytes constituting the exchange’s payload body (Section 3.3 of [RFC7230]). Note that the payload body is the message body with any transfer encodings removed.

Potentially-valid results include:

- The signed headers of the exchange so that higher-level protocols can avoid relying on unsigned headers, and
- Either a certificate chain or a public key so that a higher-level protocol can determine whether it’s actually valid.

This algorithm accepts a "forceFetch" flag that avoids the cache when fetching URLs. A client that determines that a potentially-valid certificate chain is actually invalid due to an expired OCSP response MAY retry with "forceFetch" set to retrieve an updated OCSP from the original server.

1. Let:

   * "signature" be the signature (byte sequence in the parameterised identifier’s "sig" parameter).
   * "integrity" be the signature’s "integrity" parameter.
   * "validity-url" be the signature’s "validity-url" parameter.
   * "cert-url" be the signature’s "cert-url" parameter, if any.
   * "cert-sha256" be the signature’s "cert-sha256" parameter, if any.
   * "ed25519key" be the signature’s "ed25519key" parameter, if any.
   * "date" be the signature’s "date" parameter, interpreted as a Unix time.
* "expires" be the signature’s "expires" parameter, interpreted as a Unix time.

2. Set "publicKey" and "signing-alg" depending on which key fields are present:
   1. If "cert-url" is present:
      1. Let "certificate-chain" be the result of loading the certificate chain at "cert-url" passing the "forceFetch" flag (Section 3.3). If this returns "invalid", return "invalid".
      2. Let "main-certificate" be the first certificate in "certificate-chain".
      3. Set "publicKey" to "main-certificate"’s public key.
      4. If "publicKey" is an RSA key, return "invalid".
      5. If "publicKey" is a key using the secp256r1 elliptic curve, set "signing-alg" to ecdsa_secp256r1_sha256 as defined in Section 4.2.3 of [RFC8446].
      6. Otherwise, either return "invalid" or set "signing-alg" to a non-legacy signing algorithm defined by TLS 1.3 or later ([RFC8446]). This choice MUST depend only on "publicKey”’s type and not on any other context.
   2. If "ed25519key" is present, set "publicKey" to "ed25519key" and "signing-alg" to ed25519, as defined by [RFC8032]
   3. If "expires" is more than 7 days (604800 seconds) after "date", return "invalid".
   4. If the current time is before "date" or after "expires", return "invalid".
   5. Let "message" be the concatenation of the following byte strings. This matches the [RFC8446] format to avoid cross-protocol attacks if anyone uses the same key in a TLS certificate and an exchange-signing certificate.
      1. A string that consists of octet 32 (0x20) repeated 64 times.
      2. A context string: the ASCII encoding of "HTTP Exchange 1".
Note: RFC EDITOR PLEASE DELETE THIS NOTE; The implementation of the final RFC MUST use this context string, but implementations of drafts MUST NOT use it and MUST use another draft-specific string beginning with "HTTP Exchange 1" instead. This ensures that signers can predict how their signatures will be used.

3. A single 0 byte which serves as a separator.

4. If "cert-sha256" is set, a byte holding the value 32 followed by the 32 bytes of the value of "cert-sha256". Otherwise a 0 byte.

5. The 8-byte big-endian encoding of the length in bytes of "validity-url", followed by the bytes of "validity-url".

6. The 8-byte big-endian encoding of "date".

7. The 8-byte big-endian encoding of "expires".

8. The 8-byte big-endian encoding of the length in bytes of "requestUrl", followed by the bytes of "requestUrl".

9. The 8-byte big-endian encoding of the length in bytes of "responseHeaders", followed by the bytes of "responseHeaders".

6. If "cert-url" is present and the SHA-256 hash of "main-certificate"'s "cert_data" is not equal to "cert-sha256" (whose presence was checked when the "Signature" header field was parsed), return "invalid".

Note that this intentionally differs from TLS 1.3, which signs the entire certificate chain in its Certificate Verify (Section 4.4.3 of [RFC8446]), in order to allow updating the stapled OCSP response without updating signatures at the same time.

7. If "signature" is not a valid signature of "message" by "publicKey" using "signing-alg", return "invalid".

8. If "headers", interpreted according to Section 3.2, does not contain a "Content-Type" response header field (Section 3.1.1.5 of [RFC7231]), return "invalid".

Clients MUST interpret the signed payload as this specified media type instead of trying to sniff a media type from the bytes of the payload, for example by attaching an "X-Content-
9. If "integrity" names a header field and parameter that is not present in "responseHeaders" or which the client cannot use to check the integrity of "payload" (for example, the header field is new and hasn’t been implemented yet), then return "invalid". If the selected header field provides integrity guarantees weaker than SHA-256, return "invalid". If validating integrity using the selected header field requires the client to process records larger than 16384 bytes, return "invalid". Clients MUST implement at least the "Digest" header field with its "mi-sha256" digest algorithm (Section 3 of [I-D.thomson-http-mice]).

Note: RFC EDITOR PLEASE DELETE THIS NOTE; Implementations of drafts of this RFC MUST recognize the draft spelling of the content encoding and digest algorithm specified by [I-D.thomson-http-mice] until that draft is published as an RFC. For example, implementations of draft-thomson-http-mice-03 would use "mi-sha256-03" and MUST NOT use "mi-sha256" itself. This ensures that final implementations don’t need to handle compatibility with implementations of early drafts of that content encoding.

If "payload" doesn’t match the integrity information in the header described by "integrity", return "invalid".

10. Return "potentially-valid" with whichever is present of "certificate-chain" or "ed25519key".

Note that the above algorithm can determine that an exchange’s headers are potentially-valid before the exchange’s payload is received. Similarly, if "integrity" identifies a header field and parameter like "Digest:mi-sha256" ([I-D.thomson-http-mice]) that can incrementally validate the payload, early parts of the payload can be determined to be potentially-valid before later parts of the payload. Higher-level protocols MAY process parts of the exchange that have been determined to be potentially-valid as soon as that determination is made but MUST NOT process parts of the exchange that are not yet potentially-valid. Similarly, as the higher-level protocol determines that parts of the exchange are actually valid, the client MAY process those parts of the exchange and MUST wait to process other parts of the exchange until they too are determined to be valid.
3.5.1. Open Questions

Should the signed message use the TLS format (with an initial 64 spaces) even though these certificates can’t be used in TLS servers?

3.6. Updating signature validity

Both OCSP responses and signatures are designed to expire a short time after they’re signed, so that revoked certificates and signed exchanges with known vulnerabilities are distrusted promptly.

This specification provides no way to update OCSP responses by themselves. Instead, clients need to re-fetch the "cert-url" (Section 3.5, Paragraph 6) to get a chain including a newer OCSP response.

The "validity-url" parameter (Paragraph 6) of the signatures provides a way to fetch new signatures or learn where to fetch a complete updated exchange.

Each version of a signed exchange SHOULD have its own validity URLs, since each version needs different signatures and becomes obsolete at different times.

The resource at a "validity-url" is "validity data", a CBOR map matching the following CDDL ([I-D.ietf-cbor-cddl]):

```plaintext
validity = {
  ? signatures: [ + bytes ]
  ? update: {
    ? size: uint,
  }
}
```

The elements of the "signatures" array are parameterised identifiers (Section 4.2.6 of [I-D.ietf-httpbis-header-structure]) meant to replace the signatures within the "Signature" header field pointing to this validity data. If the signed exchange contains a bug severe enough that clients need to stop using the content, the "signatures" array MUST NOT be present.

If the the "update" map is present, that indicates that a new version of the signed exchange is available at its effective request URI (Section 5.5 of [RFC7230]) and can give an estimate of the size of the updated exchange ("update.size"). If the signed exchange is currently the most recent version, the "update" SHOULD NOT be present.
If both the "signatures" and "update" fields are present, clients can use the estimated size to decide whether to update the whole resource or just its signatures.

3.6.1. Examples

For example, say a signed exchange whose URL is "https://example.com/resource" has the following "Signature" header field (with line breaks included and irrelevant fields omitted for ease of reading).

```plaintext
Signature:
sig1;
sig=*MEUCIQ...*;
...validity-url="https://example.com/resource.validity.1511157180";
cert-url="https://example.com/oldcerts";
date=1511128380; expires=1511733180,
sig2;
sig=*MEQCIG...*;
...validity-url="https://example.com/resource.validity.1511157180";
cert-url="https://example.com/newcerts";
date=1511128380; expires=1511733180,
thirdpartysig;
sig=*MEYCIQ...*;
...validity-url="https://thirdparty.example.com/resource.validity.1511161860";
cert-url="https://thirdparty.example.com/certs";
date=1511478660; expires=1511824260
```

At 2017-11-27 11:02 UTC, "sig1" and "sig2" have expired, but "thirdpartysig" doesn’t expire until 23:11 that night, so the client needs to fetch "https://example.com/resource.validity.1511157180" (the "validity-url" of "sig1" and "sig2") if it wishes to update those signatures. This URL might contain:
This indicates that the client could fetch a newer version at
"https://example.com/resource" (the original URL of the exchange), or
that the validity period of the old version can be extended by
replacing the first two of the original signatures (the ones with a
validity-url of "https://example.com/resource.validity.1511157180")
with the single new signature provided. (This might happen at the
end of a migration to a new root certificate.) The signatures of the
updated signed exchange would be:

Signature:
sig1;
sig=*MEQCIC...*;
...
validity-url="https://example.com/resource.validity.1511157180";
cert-url="https://example.com/newcerts";
date=1511733180; expires=1512337980,
thirdpartysig;
sig=*MEYCIQ...*;
...
validity-url="https://thirdparty.example.com/resource.validity.1511161860";
cert-url="https://thirdparty.example.com/certs";
date=1511478660; expires=1511824260

"https://example.com/resource.validity.1511157180" could also expand
the set of signatures if its "signatures" array contained more than 2
elements.

3.7. The Accept-Signature header

"Signature" header fields cost on the order of 300 bytes for ECDSA
signatures, so servers might prefer to avoid sending them to clients
that don’t intend to use them. A client can send the "Accept-
Signature" header field to indicate that it does intend to take
advantage of any available signatures and to indicate what kinds of signatures it supports.

When a server receives an "Accept-Signature" header field in a client request, it SHOULD reply with any available "Signature" header fields for its response that the "Accept-Signature" header field indicates the client supports. However, if the "Accept-Signature" value violates a requirement in this section, the server MUST behave as if it hadn’t received any "Accept-Signature" header at all.

The "Accept-Signature" header field is a Structured Header as defined by [I-D.ietf-httpbis-header-structure]. Its value MUST be a parameterised list (Section 3.4 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

```
Accept-Signature = sh-param-list
```

The order of identifiers in the "Accept-Signature" list is not significant. Identifiers, ignoring any initial "-" character, MUST NOT be duplicated.

Each identifier in the "Accept-Signature" header field’s value indicates that a feature of the "Signature" header field (Section 3.1) is supported. If the identifier begins with a "-" character, it instead indicates that the feature named by the rest of the identifier is not supported. Unknown identifiers and parameters MUST be ignored because new identifiers and new parameters on existing identifiers may be defined by future specifications.

### 3.7.1. Integrity identifiers

Identifiers starting with "digest/" indicate that the client supports the "Digest" header field ([RFC3230]) with the parameter from the HTTP Digest Algorithm Values Registry [6] registry named in lowercase by the rest of the identifier. For example, "digest/mi-blake2" indicates support for Merkle integrity with the as-yet-unspecified mi-blake2 parameter, and "-digest/mi-sha256" indicates non-support for Merkle integrity with the mi-sha256 content encoding.

If the "Accept-Signature" header field is present, servers SHOULD assume support for "digest/mi-sha256" unless the header field states otherwise.

### 3.7.2. Key type identifiers

Identifiers starting with "ecdsa/" indicate that the client supports certificates holding ECDSA public keys on the curve named in lowercase by the rest of the identifier.
If the "Accept-Signature" header field is present, servers SHOULD assume support for "ecdsa/secp256r1" unless the header field states otherwise.

3.7.3. Key value identifiers

The "ed25519key" identifier has parameters indicating the public keys that will be used to validate the returned signature. Each parameter’s name is re-interpreted as a byte sequence (Section 3.10 of [I-D.ietf-httpbis-header-structure]) encoding a prefix of the public key. For example, if the client will validate signatures using the public key whose base64 encoding is "11qYAYKxCr6VS/7TyWQHOG7hcvPapiMrwIaaPcHURo=", valid "Accept-Signature" header fields include:

Accept-Signature: ..., ed25519key; *11qYAYKxCr6VS/7TyWQHOG7hcvPapiMrwIaaPcHURo=*  
Accept-Signature: ..., ed25519key; *11qYAYKxCr6VS/7TyWQHOG==*
Accept-Signature: ..., ed25519key; *11qYAA==*
Accept-Signature: ..., ed25519key; **

but not

Accept-Signature: ..., ed25519key; *11qYAA==*

because 5 bytes isn’t a valid length for encoded base64, and not

Accept-Signature: ..., ed25519key; 11qYAQ

because it doesn’t start or end with the "**"s that indicate a byte sequence.

Note that "ed25519key; **" is an empty prefix, which matches all public keys, so it’s useful in subresource integrity (Appendix A.3) cases like "<link rel=preload as=script href="...">" where the public key isn’t known until the matching "<script src="..." integrity="..."></script>" tag.

3.7.4. Examples

Accept-Signature: digest/mi-sha256

states that the client will accept signatures with payload integrity assured by the "Digest" header and "mi-sha256" digest algorithm and implies that the client will accept signatures from ECDSA keys on the secp256r1 curve.

Accept-Signature: -ecdsa/secp256r1, ecdsa/secp384r1
states that the client will accept ECDSA keys on the secp384r1 curve but not the secp256r1 curve and payload integrity assured with the "Digest: mi-sha256" header field.

3.7.5. Open Questions

Is an "Accept-Signature" header useful enough to pay for itself? If clients wind up sending it on most requests, that may cost more than the cost of sending "Signature"s unconditionally. On the other hand, it gives servers an indication of which kinds of signatures are supported, which can help us upgrade the ecosystem in the future.

Is "Accept-Signature" the right spelling, or do we want to imitate "Want-Digest" (Section 4.3.1 of [RFC3230]) instead?

Do I have the right structure for the identifiers indicating feature support?

4. Cross-origin trust

To determine whether to trust a cross-origin exchange, the client takes a "Signature" header field (Section 3.1) and the exchange’s

- "requestUrl", a byte sequence that can be parsed into the exchange’s effective request URI (Section 5.5 of [RFC7230]),
- "responseHeaders", a byte sequence holding the canonical serialization (Section 3.4) of the CBOR representation (Section 3.2) of the exchange’s response metadata and headers, and
- "payload", a stream of bytes constituting the exchange’s payload body (Section 3.3 of [RFC7230]).

The client MUST parse the "Signature" header into a list of signatures according to the instructions in Section 3.5, and run the following algorithm for each signature, stopping at the first one that returns "valid". If any signature returns "valid", return "valid". Otherwise, return "invalid".

1. If the signature’s "validity-url" parameter (Paragraph 6) is not same-origin [7] with "requestUrl", return "invalid".

2. Use Section 3.5 to determine the signature’s validity for "requestUrl", "responseHeaders", and "payload", getting "certificate-chain" back. If this returned "invalid" or didn’t return a certificate chain, return "invalid".
3. Let "response" be the response metadata and headers parsed out of "responseHeaders".

4. If Section 3 of [RFC7234] forbids a shared cache from storing "response", return "invalid".

5. If "response"’s headers contain an uncached header field, as defined in Section 4.1, return "invalid".

6. Let "authority" be the host component of "requestUrl".

7. Validate the "certificate-chain" using the following substeps. If any of them fail, re-run Section 3.5 once over the signature with the "forceFetch" flag set, and restart from step 2. If a substep fails again, return "invalid".

   1. Use "certificate-chain" to validate that its first entry, "main-certificate" is trusted as "authority"’s server certificate ([RFC5280] and other undocumented conventions). Let "path" be the path that was used from the "main-certificate" to a trusted root, including the "main-certificate" but excluding the root.

   2. Validate that "main-certificate" has the CanSignHttpExchanges extension (Section 4.2).

   3. Validate that "main-certificate" has an "ocsp" property (Section 3.3) with a valid OCSP response whose lifetime ("nextUpdate - thisUpdate") is less than 7 days ([RFC6960]). Note that this does not check for revocation of intermediate certificates, and clients SHOULD implement another mechanism for that.

   4. Validate that valid SCTs from trusted logs are available from any of:

      + The "SignedCertificateTimestampList" in "main-certificate"’s "sct" property (Section 3.3),

      + An OCSP extension in the OCSP response in "main-certificate"’s "ocsp" property, or

      + An X.509 extension in the certificate in "main-certificate"’s "cert" property,

      as described by Section 3.3 of [RFC6962].

8. Return "valid".
4.1. Uncached header fields

Hop-by-hop and other uncached headers MUST NOT appear in a signed exchange. These will eventually be listed in [I-D.ietf-httpbis-cache], but for now they’re listed here:

- Hop-by-hop header fields listed in the Connection header field (Section 6.1 of [RFC7230]).
- Header fields listed in the no-cache response directive in the Cache-Control header field (Section 5.2.2.2 of [RFC7234]).
- Header fields defined as hop-by-hop:
  * Connection
  * Keep-Alive
  * Proxy-Connection
  * Trailer
  * Transfer-Encoding
  * Upgrade
- Stateful headers as defined below.

4.1.1. Stateful header fields

As described in Section 6.1, a publisher can cause problems if they sign an exchange that includes private information. There’s no way for a client to be sure an exchange does or does not include private information, but header fields that store or convey stored state in the client are a good sign.

A stateful response header field modifies state, including authentication status, in the client. The HTTP cache is not considered part of this state. These include but are not limited to:

- "Authentication-Control", [RFC8053]
- "Authentication-Info", [RFC7615]
- "Clear-Site-Data", [W3C.WD-clear-site-data-20171130]
- "Optional-WWW-Authenticate", [RFC8053]
4.2. Certificate Requirements

We define a new X.509 extension, CanSignHttpExchanges to be used in the certificate when the certificate permits the usage of signed exchanges. When this extension is not present the client MUST NOT accept a signature from the certificate as proof that a signed exchange is authoritative for a domain covered by the certificate. When it is present, the client MUST follow the validation procedure in Section 4.

    id-ce-canSignHttpExchanges OBJECT IDENTIFIER ::= { TBD }

    CanSignHttpExchanges ::= NULL

Note that this extension contains an ASN.1 NULL (bytes "05 00") because some implementations have bugs with empty extensions.

Leaf certificates without this extension need to be revoked if the private key is exposed to an unauthorized entity, but they generally don’t need to be revoked if a signing oracle is exposed and then removed.

CA certificates, by contrast, need to be revoked if an unauthorized entity is able to make even one unauthorized signature.

Certificates with this extension MUST be revoked if an unauthorized entity is able to make even one unauthorized signature.

Conforming CAs MUST NOT mark this extension as critical.
Clients MUST NOT accept certificates with this extension in TLS connections (Section 4.4.2.2 of [RFC8446]).

RFC EDITOR PLEASE DELETE THE REST OF THE PARAGRAPHS IN THIS SECTION

id-ce-google OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 11129 }
id-ce-canSignHttpExchangesDraft OBJECT IDENTIFIER ::= { id-ce-google 2 1 22 }

Implementations of drafts of this specification MAY recognize the "id-ce-canSignHttpExchangesDraft" OID as identifying the CanSignHttpExchanges extension. This OID might or might not be used as the final OID for the extension, so certificates including it might need to be reissued once the final RFC is published.

5. Transferring a signed exchange

A signed exchange can be transferred in several ways, of which three are described here.

5.1. Same-origin response

The signature for a signed exchange can be included in a normal HTTP response. Because different clients send different request header fields, clients don’t know how the server’s content negotiation algorithm works, and intermediate servers add response header fields, it can be impossible to have a signature for the exchange’s exact request, content negotiation, and response. Therefore, when a client calls the validation procedure in Section 3.5) to validate the "Signature" header field for an exchange represented as a normal HTTP request/response pair, it MUST pass:

- The "Signature" header field,
- The effective request URI (Section 5.5 of [RFC7230]) of the request,
- The serialized headers defined by Section 5.1.1, and
- The response’s payload.

If the client relies on signature validity for any aspect of its behavior, it MUST ignore any header fields that it didn’t pass to the validation procedure.

If the signed response includes a "Variants" header field, the client MUST use the cache behavior algorithm in Section 4 of [I-D.ietf-httpbis-variants] to check that the signed response is an appropriate representation for the request the client is trying to
fulfil. If the response is not an appropriate representation, the client MUST treat the signature as invalid.

5.1.1. Serialized headers for a same-origin response

The serialized headers of an exchange represented as a normal HTTP request/response pair (Section 2.1 of [RFC7230] or Section 8.1 of [RFC7540]) are the canonical serialization (Section 3.4) of the CBOR representation (Section 3.2) of the response status code (Section 6 of [RFC7231]) and the response header fields whose names are listed in that response’s "Signed-Headers" header field (Section 5.1.2). If a response header field name from "Signed-Headers" does not appear in the response’s header fields, the exchange has no serialized headers.

If the exchange’s "Signed-Headers" header field is not present, doesn’t parse as a Structured Header ([I-D.ietf-httpbis-header-structure]) or doesn’t follow the constraints on its value described in Section 5.1.2, the exchange has no serialized headers.

5.1.1.1. Open Questions

Do the serialized headers of an exchange need to include the "Signed-Headers" header field itself?

5.1.2. The Signed-Headers Header

The "Signed-Headers" header field identifies an ordered list of response header fields to include in a signature. The request URL and response status are included unconditionally. This allows a TLS-terminating intermediate to reorder headers without breaking the signature. This can also allow the intermediate to add headers that will be ignored by some higher-level protocols, but Section 3.5 provides a hook to let other higher-level protocols reject such insecure headers.

This header field appears once instead of being incorporated into the signatures’ parameters because the signed header fields need to be consistent across all signatures of an exchange, to avoid forcing higher-level protocols to merge the header field lists of valid signatures.

"Signed-Headers" is a Structured Header as defined by [I-D.ietf-httpbis-header-structure]. Its value MUST be a list (Section 3.2 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

Signed-Headers = sh-list
Each element of the "Signed-Headers" list must be a lowercase string (Section 3.8 of [I-D.ietf-httpbis-header-structure]) naming an HTTP response header field. Pseudo-header field names (Section 8.1.2.1 of [RFC7540]) MUST NOT appear in this list.

Higher-level protocols SHOULD place requirements on the minimum set of headers to include in the "Signed-Headers" header field.

5.2. HTTP/2 extension for cross-origin Server Push

To allow servers to Server-Push (Section 8.2 of [RFC7540]) signed exchanges (Section 3) signed by an authority for which the server is not authoritative (Section 9.1 of [RFC7230]), this section defines an HTTP/2 extension.

5.2.1. Indicating support for cross-origin Server Push

Clients that might accept signed Server Pushes with an authority for which the server is not authoritative indicate this using the HTTP/2 SETTINGS parameter ENABLE_CROSS_ORIGIN_PUSH (0xSETTING-TBD).

An ENABLE_CROSS_ORIGIN_PUSH value of 0 indicates that the client does not support cross-origin Push. A value of 1 indicates that the client does support cross-origin Push.

A client MUST NOT send a ENABLE_CROSS_ORIGIN_PUSH setting with a value other than 0 or 1 or a value of 0 after previously sending a value of 1. If a server receives a value that violates these rules, it MUST treat it as a connection error (Section 5.4.1 of [RFC7540]) of type PROTOCOL_ERROR.

The use of a SETTINGS parameter to opt-in to an otherwise incompatible protocol change is a use of "Extending HTTP/2" defined by Section 5.5 of [RFC7540]. If a server were to send a cross-origin Push without first receiving a ENABLE_CROSS_ORIGIN_PUSH setting with the value of 1 it would be a protocol violation.

5.2.2. NO_TRUSTED_EXCHANGE_SIGNATURE error code

The signatures on a Pushed cross-origin exchange may be untrusted for several reasons, for example that the certificate could not be fetched, that the certificate does not chain to a trusted root, that the signature itself doesn’t validate, that the signature is expired, etc. This draft conflates all of these possible failures into one error code, NO_TRUSTED_EXCHANGE_SIGNATURE (0xERROR-TBD).
5.2.2.1. Open Questions

How fine-grained should this specification’s error codes be?

5.2.3. Validating a cross-origin Push

If the client has set the ENABLE_CROSS_ORIGIN_PUSH setting to 1, the server MAY Push a signed exchange for which it is not authoritative, and the client MUST NOT treat a PUSH_PROMISE for which the server is not authoritative as a stream error (Section 5.4.2 of [RFC7540]) of type PROTOCOL_ERROR, as described in Section 8.2 of [RFC7540], unless there is another error as described below.

Instead, the client MUST validate such a PUSH_PROMISE and its response against the following list:

1. If the PUSH_PROMISE includes any non-pseudo request header fields, the client MUST treat it as a stream error (Section 5.4.2 of [RFC7540]) of type PROTOCOL_ERROR.

2. If the PUSH_PROMISE’s method is not "GET", the client MUST treat it as a stream error (Section 5.4.2 of [RFC7540]) of type PROTOCOL_ERROR.

3. Run the algorithm in Section 4 over:

   * The "Signature" header field from the response.
   * The effective request URI from the PUSH_PROMISE.
   * The canonical serialization (Section 3.4) of the CBOR representation (Section 3.2) of the pushed response’s status and its headers except for the "Signature" header field.
   * The response’s payload.

   If this returns "invalid", the client MUST treat the response as a stream error (Section 5.4.2 of [RFC7540]) of type NO_TRUSTED_EXCHANGE_SIGNATURE. Otherwise, the client MUST treat the pushed response as if the server were authoritative for the PUSH_PROMISE’s authority.

5.2.3.1. Open Questions

Is it right that "validity-url" is required to be same-origin with the exchange? This allows the mitigation against downgrades in Section 6.3, but prohibits intermediates from providing a cache of the validity information. We could do both with a list of URLs.
5.3. application/signed-exchange format

To allow signed exchanges to be the targets of "<link rel=prefetch>" tags, we define the "application/signed-exchange" content type that represents a signed HTTP exchange, including a request URL, response metadata and header fields, and a response payload.

When served over HTTP, a response containing an "application/signed-exchange" payload MUST include at least the following response header fields, to reduce content sniffing vulnerabilities (Section 6.8):

- Content-Type: application/signed-exchange;v=_version_
- X-Content-Type-Options: nosniff

This content type consists of the concatenation of the following items:

1. 8 bytes consisting of the ASCII characters "sxg1" followed by 4 0x00 bytes, to serve as a file signature. This is redundant with the MIME type, and recipients that receive both MUST check that they match and stop parsing if they don’t.

   Note: RFC EDITOR PLEASE DELETE THIS NOTE; The implementation of the final RFC MUST use this file signature, but implementations of drafts MUST NOT use it and MUST use another implementation-specific 8-byte string beginning with "sxg1-".

2. 2 bytes storing a big-endian integer "fallbackUrlLength".

3. "fallbackUrlLength" bytes holding a "fallbackUrl", which MUST be an absolute URL with a scheme of "https".

   Note: The byte location of the fallback URL is intended to remain invariant across versions of the "application/signed-exchange" format so that parsers encountering unknown versions can always find a URL to redirect to.

   Issue: Should this fallback information also include the method?

4. 3 bytes storing a big-endian integer "sigLength". If this is larger than 16384 (16*1024), parsing MUST fail.

5. 3 bytes storing a big-endian integer "headerLength". If this is larger than 524288 (512*1024), parsing MUST fail.

6. "sigLength" bytes holding the "Signature" header field’s value (Section 3.1).
7. "headerLength" bytes holding "signedHeaders", the canonical serialization (Section 3.4) of the CBOR representation of the response headers of the exchange represented by the "application/signed-exchange" resource (Section 3.2), excluding the "Signature" header field.

8. The payload body (Section 3.3 of [RFC7230]) of the exchange represented by the "application/signed-exchange" resource.

Note that the use of the payload body here means that a "Transfer-Encoding" header field inside the "application/signed-exchange" header block has no effect. A "Transfer-Encoding" header field on the outer HTTP response that transfers this resource still has its normal effect.

5.3.1. Cross-origin trust in application/signed-exchange

To determine whether to trust a cross-origin exchange stored in an "application/signed-exchange" resource, pass the "Signature" header field’s value, "fallbackUrl" as the effective request URI, "signedHeaders", and the payload body to the algorithm in Section 4.

5.3.2. Example

An example "application/signed-exchange" file representing a possible signed exchange with https://example.com/ [8] follows, with lengths represented by descriptions in "<>"s, CBOR represented in the extended diagnostic format defined in Appendix G of [I-D.ietf-cbor-cddl], and most of the "Signature" header field and payload elided with a ...:

```
xgx1\0\0\0\0<2-byte length of the following url string>
  https://example.com/<3-byte length of the following header value>
<3-byte length of the encoding of the following map>sig1; sig=*...; integrity="digest/mi-sha256"; {...
  ':status': '200',
  'content-type': 'text/html'
}
<!doctype html>
<html>...
```

5.3.3. Open Questions

Should this be a CBOR format, or is the current mix of binary and CBOR better?

Are the mime type, extension, and magic number right?
6. Security considerations

6.1. Over-signing

If a publisher blindly signs all responses as their origin, they can cause at least two kinds of problems, described below. To avoid this, publishers SHOULD design their systems to opt particular public content that doesn’t depend on authentication status into signatures instead of signing by default.

Signing systems SHOULD also incorporate the following mitigations to reduce the risk that private responses are signed:

1. Strip the "Cookie" request header field and other identifying information like client authentication and TLS session IDs from requests whose exchange is destined to be signed, before forwarding the request to a backend.

2. Only sign exchanges where the response includes a "Cache-Control: public" header. Clients are not required to fail signature-checking for exchanges that omit this "Cache-Control" response header field to reduce the risk that naive signing systems blindly add it.

6.1.1. Session fixation

Blind signing can sign responses that create session cookies or otherwise change state on the client to identify a particular session. This breaks certain kinds of CSRF defense and can allow an attacker to force a user into the attacker’s account, where the user might unintentionally save private information, like credit card numbers or addresses.

This specification defends against cookie-based attacks by blocking the "Set-Cookie" response header, but it cannot prevent Javascript or other response content from changing state.

6.1.2. Misleading content

If a site signs private information, an attacker might set up their own account to show particular private information, forward that signed information to a victim, and use that victim’s confusion in a more sophisticated attack.

Stripping authentication information from requests before sending them to backends is likely to prevent the backend from showing attacker-specific information in the signed response. It does not prevent the attacker from showing their victim a signed-out page when
the victim is actually signed in, but while this is still misleading, it seems less likely to be useful to the attacker.

6.2. Off-path attackers

Relaxing the requirement to consult DNS when determining authority for an origin means that an attacker who possesses a valid certificate no longer needs to be on-path to redirect traffic to them; instead of modifying DNS, they need only convince the user to visit another Web site in order to serve responses signed as the target. This consideration and mitigations for it are shared by the combination of [RFC8336] and [I-D.ietf-httpbis-http2-secondary-certs].

6.3. Downgrades

Signing a bad response can affect more users than simply serving a bad response, since a served response will only affect users who make a request while the bad version is live, while an attacker can forward a signed response until its signature expires. Publishers should consider shorter signature expiration times than they use for cache expiration times.

Clients MAY also check the "validity-url" (Paragraph 6) of an exchange more often than the signature’s expiration would require. Doing so for an exchange with an HTTPS request URI provides a TLS guarantee that the exchange isn’t out of date (as long as Section 5.2.3.1 is resolved to keep the same-origin requirement).

6.4. Signing oracles are permanent

An attacker with temporary access to a signing oracle can sign "still valid" assertions with arbitrary timestamps and expiration times. As a result, when a signing oracle is removed, the keys it provided access to MUST be revoked so that, even if the attacker used them to sign future-dated exchange validity assertions, the key’s OCSP assertion will expire, causing the exchange as a whole to become untrusted.

6.5. Unsigned headers

The use of a single "Signed-Headers" header field prevents us from signing aspects of the request other than its effective request URI (Section 5.5 of [RFC7230]). For example, if a publisher signs both "Content-Encoding: br" and "Content-Encoding: gzip" variants of a response, what’s the impact if an attacker serves the brotli one for a request with "Accept-Encoding: gzip"? This is mitigated by using
[I-D.ietf-httpbis-variants] instead of request headers to describe how the client should run content negotiation.

The simple form of "Signed-Headers" also prevents us from signing less than the full request URL. The SRI use case (Appendix A.3) may benefit from being able to leave the authority less constrained.

Section 3.5 can succeed when some delivered headers aren't included in the signed set. This accommodates current TLS-terminating intermediates and may be useful for SRI (Appendix A.3), but is risky for trusting cross-origin responses (Appendix A.1, Appendix A.2, and Appendix A.6). Section 5.2 requires all headers to be included in the signature before trusting cross-origin pushed resources, at Ryan Sleevi's recommendation.

6.6. application/signed-exchange

Clients MUST NOT trust an effective request URI claimed by an "application/signed-exchange" resource (Section 5.3) without either ensuring the resource was transferred from a server that was authoritative (Section 9.1 of [RFC7230]) for that URI’s origin, or calling the algorithm in Section 5.3.1 and getting "valid" back.

6.7. Key re-use with TLS

In general, key re-use across multiple protocols is a bad idea.

Using an exchange-signing key in a TLS (or other directly-internet-facing) server increases the risk that an attacker can steal the private key, which will allow them to mint packages (similar to Section 6.4) until their theft is discovered.

Using a TLS key in a CanSignHttpExchanges certificate makes it less likely that the server operator will discover key theft, due to the considerations in Section 6.2.

This specification uses the CanSignHttpExchanges X.509 extension (Section 4.2) to discourage re-use of TLS keys to sign exchanges or vice-versa.

We require that clients reject certificates with the CanSignHttpExchanges extension when making TLS connections to minimize the chance that servers will re-use keys like this. Ideally, we would make the extension critical so that even clients that don’t understand it would reject such TLS connections, but this proved impossible because certificate-validating libraries ship on significantly different schedules from the clients that use them.
Even once all clients reject these certificates in TLS connections, this will still just discourage and not prevent key re-use, since a server operator can unwisely request two different certificates with the same private key.

6.8. Content sniffing

While modern browsers tend to trust the "Content-Type" header sent with a resource, especially when accompanied by "X-Content-Type-Options: nosniff", plugins will sometimes search for executable content buried inside a resource and execute it in the context of the origin that served the resource, leading to XSS vulnerabilities. For example, some PDF reader plugins look for "%PDF" anywhere in the first 1kB and execute the code that follows it.

The "application/signed-exchange" format (Section 5.3) includes a URL and response headers early in the format, which an attacker could use to cause these plugins to sniff a bad content type.

To avoid vulnerabilities, in addition to the response header requirements in Section 5.3, servers are advised to only serve an "application/signed-exchange" resource (SXG) from a domain if it would also be safe for that domain to serve the SXG’s content directly, and to follow at least one of the following strategies:

1. Only serve signed exchanges from dedicated domains that don’t have access to sensitive cookies or user storage.

2. Generate signed exchanges "offline", that is, in response to a trusted author submitting content or existing signatures reaching a certain age, rather than in response to untrusted-reader queries.

3. Do all of:

   1. If the SXG’s fallback URL (Section 5.3) is derived from the request URL, percent-encode [9] ([URL]) any bytes that are greater than 0x7E or are not URL code points [10] ([URL]) in the fallback URL. It is particularly important to make sure no unescaped nulls (0x00) or angle brackets (0x3C and 0x3E) appear.

   2. Do not reflect request header fields into the set of response headers.

There are still a few binary length fields that an attacker may influence to contain sensitive bytes, but they’re always followed by lowercase alphabetic strings from a small set of possibilities, which
reduces the chance that a client will sniff them as indicating a particular content type.

To encourage servers to include the "X-Content-Type-Options: nosniff" header field, clients SHOULD reject signed exchanges served without it.

7. Privacy considerations

Normally, when a client fetches "https://o1.com/resource.js", "o1.com" learns that the client is interested in the resource. If "o1.com" signs "resource.js", "o2.com" serves it as "https://o2.com/o1resource.js", and the client fetches it from there, then "o2.com" learns that the client is interested, and if the client executes the Javascript, that could also report the client’s interest back to "o1.com".

Often, "o2.com" already knew about the client’s interest, because it’s the entity that directed the client to "o1resource.js", but there may be cases where this leaks extra information.

For non-executable resource types, a signed response can improve the privacy situation by hiding the client’s interest from the original publisher.

To prevent network operators other than "o1.com" or "o2.com" from learning which exchanges were read, clients SHOULD only load exchanges fetched over a transport that’s protected from eavesdroppers. This can be difficult to determine when the exchange is being loaded from local disk, but when the client itself requested the exchange over a network it SHOULD require TLS ([RFC8446]) or a successor transport layer, and MUST NOT accept exchanges transferred over plain HTTP without TLS.

8. IANA considerations

TODO: possibly register the validity-url format.

8.1. Signature Header Field Registration

This section registers the "Signature" header field in the "Permanent Message Header Field Names" registry ([RFC3864]).

Header field name: "Signature"

Applicable protocol: http

Status: standard
8.2. Accept-Signature Header Field Registration

This section registers the "Accept-Signature" header field in the "Permanent Message Header Field Names" registry ([RFC3864]).

Header field name: "Accept-Signature"

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.1 of this document

8.3. Signed-Headers Header Field Registration

This section registers the "Signed-Headers" header field in the "Permanent Message Header Field Names" registry ([RFC3864]).

Header field name: "Signed-Headers"

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.7 of this document

8.4. HTTP/2 Settings

This section establishes an entry for the HTTP/2 Settings Registry that was established by Section 11.3 of [RFC7540]

Name: ENABLE_CROSS_ORIGIN_PUSH

Code: 0xSETTING-TBD

Initial Value: 0

Specification: This document
8.5. HTTP/2 Error code

This section establishes an entry for the HTTP/2 Error Code Registry that was established by Section 11.4 of [RFC7540]

Name: NO_TRUSTED_EXCHANGE_SIGNATURE

Code: 0xERROR-TBD

Description: The client does not trust the signature for a cross-origin Pushed signed exchange.

Specification: This document

8.6. Internet Media Type application/signed-exchange

Type name: application

Subtype name: signed-exchange

Required parameters:

- v: A string denoting the version of the file format. ([RFC5234] ABNF: "version = DIGIT/%x61-7A") The version defined in this specification is "1". When used with the "Accept" header field (Section 5.3.1 of [RFC7231]), this parameter can be a comma (,)-separated list of version strings. ([RFC5234] ABNF: "version-list = version *("," version )") The server is then expected to reply with a resource using a particular version from that list.

  Note: RFC EDITOR PLEASE DELETE THIS NOTE; Implementations of drafts of this specification MUST NOT use simple integers to describe their versions, and MUST instead define implementation-specific strings to identify which draft is implemented. The newest version of [I-D.yasskin-httpbis-origin-signed-exchanges-impl] describes the meaning of one such string.

Optional parameters: N/A

Encoding considerations: binary

Security considerations: see Section 6.6

Interoperability considerations: N/A

Published specification: This specification (see Section 5.3).
Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): 73 78 67 31 00

File extension(s): .sxg

Macintosh file type code(s): N/A

Person and email address to contact for further information: See Authors’ Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors’ Addresses section.

Change controller: IESG

8.7. Internet Media Type application/cert-chain+cbor

Type name: application

Subtype name: cert-chain+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: N/A

Interoperability considerations: N/A

Published specification: This specification (see Section 3.3).

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:
Deprecated alias names for this type: N/A

Magic number(s): 1*9(??) 67 F0 9F 93 9C E2 9B 93

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: See Authors’ Addresses section.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See Authors’ Addresses section.

Change controller: IESG

9. References

9.1. Normative References


9.2. Informative References

[I-D.burke-content-signature]
Burke, B., "HTTP Header for digital signatures", draft-burke-content-signature-00 (work in progress), March 2011.

[I-D.cavage-http-signatures]
[I-D.ietf-httpbis-cache]

[I-D.ietf-httpbis-http2-secondary-certs]

[I-D.thomson-http-content-signature]

[I-D.yasskin-httpbis-origin-signed-exchanges-impl]

[I-D.yasskin-webpackage-use-cases]
Yasskin, J., "Use Cases and Requirements for Web Packages", draft-yasskin-webpackage-use-cases-01 (work in progress), March 2018.


9.3. URIs

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


[8] https://example.com/


[16] https://github.com/WICG/webpackage

[17] https://www.imperialviolet.org/2012/02/05/crlsets.html


Appendix A. Use cases

A.1. PUSHed subresources

To reduce round trips, a server might use HTTP/2 Push (Section 8.2 of [RFC7540]) to inject a subresource from another server into the client’s cache. If anything about the subresource is expired or
can’t be verified, the client would fetch it from the original server.

For example, if "https://example.com/index.html" includes

<script src="https://jquery.com/jquery-1.2.3.min.js">

Then to avoid the need to look up and connect to "jquery.com" in the critical path, "example.com" might push that resource signed by "jquery.com".

A.2. Explicit use of a content distributor for subresources

In order to speed up loading but still maintain control over its content, an HTML page in a particular origin "O.com" could tell clients to load its subresources from an intermediate content distributor that’s not authoritative, but require that those resources be signed by "O.com" so that the distributor couldn’t modify the resources. This is more constrained than the common CDN case where "O.com" has a CNAME granting the CDN the right to serve arbitrary content as "O.com".

<img logicalsrc="https://O.com/img.png"
     physicalsrc="https://distributor.com/O.com/img.png">

To make it easier to configure the right distributor for a given request, computation of the "physicalsrc" could be encapsulated in a custom element:

<dist-img src="https://O.com/img.png"></dist-img>

where the "<dist-img>" implementation generates an appropriate "<img>" based on, for example, a "<meta name="dist-base">" tag elsewhere in the page. However, this has the downside that the preloader [11] can no longer see the physical source to download it. The resulting delay might cancel out the benefit of using a distributor.

This could be used for some of the same purposes as SRI (Appendix A.3).

To implement this with the current proposal, the distributor would respond to the physical request to "https://distributor.com/O.com/img.png" with first a signed PUSH_PROMISE for "https://O.com/img.png" and then a redirect to "https://O.com/img.png".
A.3. Subresource Integrity

The W3C WebAppSec group is investigating using signatures [12] in [SRI]. They need a way to transmit the signature with the response, which this proposal provides.

Their needs are simpler than most other use cases in that the "integrity="ed25519-[public-key]"" attribute and CSP-based ways of expressing a public key don’t need that key to be wrapped into a certificate.

The "ed25519key" signature parameter supports this simpler way of attaching a key.

The current proposal for signature-based SRI describes signing only the content of a resource, while this specification requires them to sign the request URI as well. This issue is tracked in https://github.com/mikewest/signature-based-sri/issues/5 [13]. The details of what they need to sign will affect whether and how they can use this proposal.

A.4. Binary Transparency

So-called "Binary Transparency" may eventually allow users to verify that a program they’ve been delivered is one that’s available to the public, and not a specially-built version intended to attack just them. Binary transparency systems don’t exist yet, but they’re likely to work similarly to the successful Certificate Transparency logs described by [RFC6962].

Certificate Transparency depends on Signed Certificate Timestamps that prove a log contained a particular certificate at a particular time. To build the same thing for Binary Transparency logs containing HTTP resources or full websites, we’ll need a way to provide signatures of those resources, which signed exchanges provides.

A.5. Static Analysis

Native app stores like the Apple App Store [14] and the Android Play Store [15] grant their contents powerful abilities, which they attempt to make safe by analyzing the applications before offering them to people. The web has no equivalent way for people to wait to run an update of a web application until a trusted authority has vouched for it.

While full application analysis probably needs to wait until the authority can sign bundles of exchanges, authorities may be able to
guarantee certain properties by just checking a top-level resource and its [SRI]-constrained sub-resources.

A.6. Offline websites

Fully-offline websites can be represented as bundles of signed exchanges, although an optimization to reduce the number of signature verifications may be needed. Work on this is in progress in the https://github.com/WICG/webpackage [16] repository.

Appendix B. Requirements

B.1. Proof of origin

To verify that a thing came from a particular origin, for use in the same context as a TLS connection, we need someone to vouch for the signing key with as much verification as the signing keys used in TLS. The obvious way to do this is to re-use the web PKI and CA ecosystem.

B.1.1. Certificate constraints

If we re-use existing TLS server certificates, we incur the risks that:

1. TLS server certificates must be accessible from online servers, so they’re easier to steal or use as signing oracles than an offline key. An exchange’s signing key doesn’t need to be online.

2. A server using an origin-trusted key for one purpose (e.g. TLS) might accidentally sign something that looks like an exchange, or vice versa.

These risks are considered too high, so we define a new X.509 certificate extension in Section 4.2 that requires CAs to issue new certificates for this purpose. We expect at least one low-cost CA to be willing to sign certificates with this extension.

B.1.2. Signature constraints

In order to prevent an attacker who can convince the server to sign some resource from causing those signed bytes to be interpreted as something else the new X.509 extension here is forbidden from being used in TLS servers. If Section 4.2 changes to allow re-use in TLS servers, we would need to:
1. Avoid key types that are used for non-TLS protocols whose output could be confused with a signature. That may be just the "rsaEncryption" OID from [RFC8017].

2. Use the same format as TLS’s signatures, specified in Section 4.4.3 of [RFC8446], with a context string that’s specific to this use.

The specification also needs to define which signing algorithm to use. It currently specifies that as a function from the key type, instead of allowing attacker-controlled data to specify it.

B.1.3. Retrieving the certificate

The client needs to be able to find the certificate vouching for the signing key, a chain from that certificate to a trusted root, and possibly other trust information like SCTs ([RFC6962]). One approach would be to include the certificate and its chain in the signature metadata itself, but this wastes bytes when the same certificate is used for multiple HTTP responses. If we decide to put the signature in an HTTP header, certificates are also unusually large for that context.

Another option is to pass a URL that the client can fetch to retrieve the certificate and chain. To avoid extra round trips in fetching that URL, it could be bundled (Appendix A.6) with the signed content or PUSHed (Appendix A.1) with it. The risks from the "client_certificate_url" extension (Section 11.3 of [RFC6066]) don’t seem to apply here, since an attacker who can get a client to load an exchange and fetch the certificates it references, can also get the client to perform those fetches by loading other HTML.

To avoid using an unintended certificate with the same public key as the intended one, the content of the leaf certificate or the chain should be included in the signed data, like TLS does (Section 4.4.3 of [RFC8446]).

B.2. How much to sign

The previous [I-D.thomson-http-content-signature] and [I-D.burke-content-signature] schemes signed just the content, while ([I-D.cavage-http-signatures] could also sign the response headers and the request method and path. However, the same path, response headers, and content may mean something very different when retrieved from a different server. Section 5.1.1 currently includes the whole request URL in the signature, but it’s possible we need a more flexible scheme to allow some higher-level protocols to accept a less-signed URL.
Servers might want to sign other request headers in order to capture their effects on content negotiation. However, there’s no standard algorithm to check that a client’s actual request headers match request headers sent by a server. The most promising attempt at this is [I-D.ietf-httpbis-variants], which encodes the content negotiation algorithm into the "Variants" and "Variant-Key" response headers. The proposal here (Section 3) assumes that is in use and doesn’t sign request headers.

B.2.1. Conveying the signed headers

HTTP headers are traditionally munged by proxies, making it impossible to guarantee that the client will see the same sequence of bytes as the publisher published. In the HTTPS world, we have more end-to-end header integrity, but it’s still likely that there are enough TLS-terminating proxies that the publisher’s signatures would tend to break before getting to the client.

There’s no way in current HTTP for the response to a client-initiated request (Section 8.1 of [RFC7540]) to convey the request headers it expected to respond to, but we sidestep that by conveying content negotiation information in response headers, per [I-D.ietf-httpbis-variants].

Since proxies are unlikely to modify unknown content types, we can wrap the original exchange into an "application/signed-exchange" format (Section 5.3) and include the "Cache-Control: no-transform" header when sending it.

To reduce the likelihood of accidental modification by proxies, the "application/signed-exchange" format includes a file signature that doesn’t collide with other known signatures.

To help the PUSHed subresources use case (Appendix A.1), we might also want to extend the "PUSH_PROMISE" frame type to include a signature, and that could tell intermediates not to change the ensuing headers.

B.3. Response lifespan

A normal HTTPS response is authoritative only for one client, for as long as its cache headers say it should live. A signed exchange can be re-used for many clients, and if it was generated while a server was compromised, it can continue compromising clients even if their requests happen after the server recovers. This signing scheme needs to mitigate that risk.
B.3.1. Certificate revocation

Certificates are mis-issued and private keys are stolen, and in response clients need to be able to stop trusting these certificates as promptly as possible. Online revocation checks don’t work [17], so the industry has moved to pushed revocation lists and stapled OCSP responses [RFC6066].

Pushed revocation lists work as-is to block trust in the certificate signing an exchange, but the signatures need an explicit strategy to staple OCSP responses. One option is to extend the certificate download (Appendix B.1.3) to include the OCSP response too, perhaps in the TLS 1.3 CertificateEntry [18] format.

B.3.2. Response downgrade attacks

The signed content in a response might be vulnerable to attacks, such as XSS, or might simply be discovered to be incorrect after publication. Once the author fixes those vulnerabilities or mistakes, clients should stop trusting the old signed content in a reasonable amount of time. Similar to certificate revocation, I expect the best option to be stapled "this version is still valid" assertions with short expiration times.

These assertions could be structured as:

1. A signed minimum version number or timestamp for a set of request headers: This requires that signed responses need to include a version number or timestamp, but allows a server to provide a single signature covering all valid versions.

2. A replacement for the whole exchange’s signature. This requires the publisher to separately re-sign each valid version and requires each version to include a different update URL, but allows intermediates to serve less data. This is the approach taken in Section 3.

3. A replacement for the exchange’s signature and an update for the embedded "expires" and related cache-control HTTP headers [RFC7234]. This naturally extends publishers’ intuitions about cache expiration and the existing cache revalidation behavior to signed exchanges. This is sketched and its downsides explored in Appendix C.

The signature also needs to include instructions to intermediates for how to fetch updated validity assertions.
B.4. Low implementation complexity

Simpler implementations are, all things equal, less likely to include bugs. This section describes decisions that were made in the rest of the specification to reduce complexity.

B.4.1. Limited choices

In general, we’re trying to eliminate unnecessary choices in the specification. For example, instead of requiring clients to support two methods for verifying payload integrity, we only require one.

B.4.2. Bounded-buffering integrity checking

Clients can be designed with a more-trusted network layer that decides how to trust resources and then provides those resources to less-trusted rendering processes along with handles to the storage and other resources they’re allowed to access. If the network layer can enforce that it only operates on chunks of data up to a certain size, it can avoid the complexity of spooling large files to disk.

To allow the network layer to verify signed exchanges using a bounded amount of memory, Section 5.3 requires the signature to be less than 16kB and the headers to be less than 512kB, and Section 3.5 requires that the MI record size be less than 16kB. This allows the network layer to validate a bounded chunk at a time, and pass that chunk on to a renderer, and then forget about that chunk before processing the next one.

The "Digest" header field from [RFC3230] requires the network layer to buffer the entire response body, so it’s disallowed.

Appendix C. Determining validity using cache control

This draft could expire signature validity using the normal HTTP cache control headers ([RFC7234]) instead of embedding an expiration date in the signature itself. This section specifies how that would work, and describes why I haven't chosen that option.

The signatures in the "Signature" header field (Section 3.1) would no longer contain "date" or "expires" fields.

The validity-checking algorithm (Section 3.5) would initialize "date" from the resource’s "Date" header field (Section 7.1.1.2 of [RFC7231]) and initialize "expires" from either the "Expires" header field (Section 5.3 of [RFC7234]) or the "Cache-Control" header field’s "max-age" directive (Section 5.2.2.8 of [RFC7234]) (added to
"date"), whichever is present, preferring "max-age" (or failing) if both are present.

Validity updates (Section 3.6) would include a list of replacement response header fields. For each header field name in this list, the client would remove matching header fields from the stored exchange’s response header fields. Then the client would append the replacement header fields to the stored exchange’s response header fields.

C.1. Example of updating cache control

For example, given a stored exchange of:

GET / HTTP/1.1
Host: example.com
Accept: */*

HTTP/1.1 200
Date: Mon, 20 Nov 2017 10:00:00 UTC
Content-Type: text/html
Date: Tue, 21 Nov 2017 10:00:00 UTC
Expires: Sun, 26 Nov 2017 10:00:00 UTC

<!doctype html>
<html>
...
</html>

And an update listing the following headers:

Expires: Fri, 1 Dec 2017 10:00:00 UTC
Date: Sat, 25 Nov 2017 10:00:00 UTC

The resulting stored exchange would be:

GET / HTTP/1.1
Host: example.com
Accept: */*

HTTP/1.1 200
Content-Type: text/html
Expires: Fri, 1 Dec 2017 10:00:00 UTC
Date: Sat, 25 Nov 2017 10:00:00 UTC

<!doctype html>
<html>
...

...
C.2. Downsides of updating cache control

In an exchange with multiple signatures, using cache control to expire signatures forces all signatures to initially live for the same period. Worse, the update from one signature's "validity-url" might not match the update for another signature. Clients would need to maintain a current set of headers for each signature, and then decide which set to use when actually parsing the resource itself.

This need to store and reconcile multiple sets of headers for a single signed exchange argues for embedding a signature's lifetime into the signature.

Appendix D. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-05

- Define absolute URLs, and limit the schemes each instance can use.
- Fill in TBD size limits.
- Update to mice-03 including the Digest header.
- Refer to draft-yasskin-httpbis-origin-signed-exchanges-impl for draft version numbers.
- Require "exchange"’s response to be cachable by a shared cache.
- Define the "integrity" field of the Signature header to include subfields of the main integrity-protecting header, including the digest algorithm.
- Put a fallback URL at the beginning of the "application/signed-exchange" format, which replaces the ':url' key from the CBOR representation of the exchange's request and response metadata and headers.
- Remove the rest of the request headers from the signed data, in favor of representing content negotiation with the "Variants" response header.
- Make the signed message format a concatenation of byte sequences, which helps implementations avoid re-serializing the exchange’s request and response metadata and headers.
- Explicitly check the response payload’s integrity instead of assuming the client did it elsewhere in processing the response.
- Reject uncached header fields.
- Update to draft-ietf-httpbis-header-structure-09.
- Update to the final TLS 1.3 RFC.
  
  draft-04
- Update to draft-ietf-httpbis-header-structure-06.
- Replace the application/http-exchange+cbor format with a simpler application/signed-exchange format that:
  - Doesn’t require a streaming CBOR parser parse it from a network stream.
  - Doesn’t allow request payloads or response trailers, which don’t fit into the signature model.
  - Allows checking the signature before parsing the exchange headers.
- Require absolute URLs.
- Make all identifiers in headers lower-case, as required by Structured Headers.
- Switch back to the TLS 1.3 signature format.
- Include the version and draft number in the signature context string.
- Remove support for integrity protection using the Digest header field.
- Limit the record size in the mi-sha256 encoding.
- Forbid RSA keys, and only require clients to support secp256r1 keys.
- Add a test OID for the CanSignHttpExchanges X.509 extension.
  
  draft-03
Allow each method of transferring an exchange to define which headers are signed, have the cross-origin methods use all headers, and remove the "allResponseHeaders" flag.

Describe footguns around signing private content, and block certain headers to make it less likely.

Define a CBOR structure to hold the certificate chain instead of re-using the TLS1.3 message. The TLS 1.3 parser fails on unexpected extensions while this format should ignore them, and apparently TLS implementations don’t expose their message parsers enough to allow passing a message to a certificate verifier.

Require an X.509 extension for the signing certificate.

draft-02

Signatures identify a header (e.g. Digest or MI) to guard the payload’s integrity instead of directly signing over the payload.

The validityUrl is signed.

Use CBOR maps where appropriate, and define how they’re canonicalized.

Remove the update.url field from signature validity updates, in favor of just re-fetching the original request URL.

Define an HTTP/2 extension to use a setting to enable cross-origin Server Push.

Define an "Accept-Signature" header to negotiate whether to send Signatures and which ones.

Define an "application/http-exchange+cbor" format to fetch signed exchanges without HTTP/2 Push.

2 new use cases.

Appendix E. Acknowledgements

Thanks to Devin Mullins, Ilari Liusvaara, Justin Schuh, Mark Nottingham, Mike Bishop, Ryan Sleevi, and Yoav Weiss for comments that improved this draft.