

HTTP  
Internet-Draft  
Obsoletes: 3205 (if approved)  
Intended status: Best Current Practice  
Expires: 28 February 2022

M. Nottingham  
27 August 2021

Building Protocols with HTTP  
draft-ietf-httpbis-bcp56bis-15

Abstract

Applications often use HTTP as a substrate to create HTTP-based APIs. This document specifies best practices for writing specifications that use HTTP to define new application protocols. 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.

This document obsoletes [RFC3205].

Note to Readers

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

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

Working Group information can be found at <http://httpwg.github.io/> (<http://httpwg.github.io/>); source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/bcp56bis> (<https://github.com/httpwg/http-extensions/labels/bcp56bis>).

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 28 February 2022.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	3
1.1. Notational Conventions . . . . .	5
2. Is HTTP Being Used? . . . . .	5
2.1. Non-HTTP Protocols . . . . .	5
3. What's Important About HTTP . . . . .	6
3.1. Generic Semantics . . . . .	6
3.2. Links . . . . .	7
3.3. Rich Functionality . . . . .	7
4. Best Practices for Specifying the Use of HTTP . . . . .	8
4.1. Specifying the Use of HTTP . . . . .	8
4.2. Specifying Server Behaviour . . . . .	9
4.3. Specifying Client Behaviour . . . . .	10
4.4. Specifying URLs . . . . .	11
4.4.1. Discovering an Application's URLs . . . . .	11
4.4.2. Considering URI Schemes . . . . .	12
4.4.3. Transport Ports . . . . .	13
4.5. Using HTTP Methods . . . . .	14
4.5.1. GET . . . . .	14
4.5.2. OPTIONS . . . . .	15
4.6. Using HTTP Status Codes . . . . .	16
4.6.1. Redirection . . . . .	17
4.7. Specifying HTTP Header Fields . . . . .	18
4.8. Defining Message Content . . . . .	20
4.9. Leveraging HTTP Caching . . . . .	20
4.9.1. Freshness . . . . .	20

4.9.2. Stale Responses . . . . .	21
4.9.3. Caching and Application Semantics . . . . .	21
4.9.4. Varying Content Based Upon the Request . . . . .	22
4.10. Handling Application State . . . . .	22
4.11. Making Multiple Requests . . . . .	22
4.12. Client Authentication . . . . .	23
4.13. Co-Existing with Web Browsing . . . . .	24
4.14. Maintaining Application Boundaries . . . . .	25
4.15. Using Server Push . . . . .	26
4.16. Allowing Versioning and Evolution . . . . .	27
5. IANA Considerations . . . . .	27
6. Security Considerations . . . . .	27
6.1. Privacy Considerations . . . . .	28
7. References . . . . .	29
7.1. Normative References . . . . .	29
7.2. Informative References . . . . .	30
Appendix A. Changes from RFC 3205 . . . . .	33
Author's Address . . . . .	33

## 1. Introduction

Applications other than Web browsing often use HTTP [HTTP] as a substrate, a practice sometimes referred to as creating "HTTP-based APIs", "REST 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, intended for only deployment by one or a few servers and consumption by a limited set of clients. As a result, 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, is deployed on multiple uncoordinated servers, and is consumed by diverse clients -- as is often the case for HTTP APIs defined by standards efforts -- tools and practices intended for limited deployment can become unsuitable.

This mismatch is largely because the API's clients and servers will implement and evolve at different paces, leading to a need for deployments with different features and versions to co-exist. As a result, the designers of HTTP-based APIs intended for such deployments need to more carefully consider how extensibility of the service will be handled and how different deployment requirements will be accommodated.

More generally, an application protocol using HTTP faces a number of design decisions, including:

- \* Should it define a new URI 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 for the specification of such applications. Section 2 defines when it applies; Section 3 surveys the properties of HTTP that are important to preserve, and Section 4 conveys best practices for specifying them.

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.

This document obsoletes [RFC3205], to reflect experience and developments regarding HTTP in the intervening time.

### 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 recommendations in this document apply when a specification defines an application that:

- \* uses the transport port 80 or 443, or
- \* uses the URI scheme "http" or "https", or
- \* uses an ALPN protocol ID [RFC7301] that generically identifies HTTP (e.g., "http/1.1", "h2", "h3"), or
- \* makes registrations in or overall modifications to the IANA registries defined for HTTP.

Additionally, when a specification is using HTTP, all of the requirements of the HTTP protocol suite are in force (in particular, [HTTP], but also other specifications such as the specific version of HTTP in use, and any extensions in use).

Note that this document is intended to apply to applications, not generic extensions to HTTP. Furthermore, while it is intended for IETF-specified applications, other standards organisations are encouraged to adhere to its requirements.

### 2.1. Non-HTTP Protocols

An application can rely upon HTTP without meeting the criteria for using it defined above. For example, an application might wish to avoid re-specifying parts of the message format, but change other aspects of the protocol's operation; or, it might want to use application-specific methods.

Doing so brings more freedom to modify protocol operations, but loses at least a portion of the benefits outlined in Section 3, as most HTTP implementations won't be easily adaptable to these changes, and the benefit of mindshare will be lost.

Such specifications MUST NOT use HTTP's URI schemes, transport ports, ALPN protocol IDs or IANA registries; rather, they are encouraged to establish their own.

### 3. What's Important About HTTP

This section examines the characteristics of HTTP that are important to consider when using HTTP to define an application protocol.

#### 3.1. Generic Semantics

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 best expressed in message content and in header fields, not status codes or methods (although the latter do have generic semantics that relate to application state).

This generic/application-specific split allows a HTTP message to be handled by common software (e.g., HTTP servers, intermediaries, client implementations, and caches) without understanding the specific application. It also allows people to leverage their knowledge of HTTP semantics without specialising them for a particular application.

Therefore, applications that use HTTP MUST NOT re-define, refine or overlay the semantics of generic protocol elements such as methods, status codes or existing header fields. Instead, they should focus their specifications on protocol elements that are specific to that application; namely their HTTP resources.

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; for example, there might be a proxy that requires authentication, or a server-side error, or a redirection. If the client does not anticipate this, the application's deployment is brittle.

See Section 4.2 for more 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 [RFC8820], 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 URI components like paths, it is RECOMMENDED that applications using HTTP define and use links [WEB-LINKING] 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 [HTTP2] and HTTP/3 [HTTP3])
- \* 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 Specifying the Use of HTTP

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

##### 4.1. Specifying the Use of HTTP

Specifications should use [HTTP] as the primary reference for HTTP; 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).

Because HTTP is a hop-by-hop protocol, a 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. Therefore, it is NOT RECOMMENDED that applications using HTTP specify a minimum version of HTTP to be used.

However, if an application's deployment would benefit from the use of a particular version of HTTP (for example, HTTP/2's multiplexing), this ought 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 complete header sections, preferably in HTTP/1.1 format [HTTP11]. 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
```

[content here]

#### 4.2. Specifying Server Behaviour

The server-side behaviours of an application are most effectively specified by defining the following protocol elements:

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

An application can define its operation by composing these protocol elements to define a set of resources that are identified by link relations and 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 content format;
- \* data processing using POST and identified request and response content 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.

Applications can also specify the use of URI Templates [URI-TEMPLATE] to allow clients to generate URLs based upon runtime data.

#### 4.3. Specifying Client Behaviour

An application's expectations for client behaviour ought to be closely aligned with those of Web browsers, to avoid interoperability issues when they are used.

One way to do this is to define it in terms of [FETCH], since that is the abstraction that browsers use for HTTP.

Some client behaviours (e.g., automatic redirect handling) and extensions (e.g., Cookies) are not required by HTTP, but nevertheless have become very common. If their use is not explicitly specified by applications using HTTP, there may be confusion and interoperability problems. In particular:

- \* Redirect handling - Applications need to specify how redirects are expected to be handled; see Section 4.6.1.
- \* Cookies - Applications using HTTP should explicitly reference the Cookie specification [COOKIES] if they are required.
- \* Certificates - Applications using HTTP should specify that TLS certificates are to be checked according to Section 4.3.4 of [HTTP] when HTTPS is used.

Applications using HTTP should not statically require HTTP features that are usually negotiated to be supported by clients. For example, requiring that clients support responses with a certain content-coding ([HTTP], Section 8.4.1) instead of negotiating for it ([HTTP], Section 12.5.3) means that otherwise conformant clients cannot interoperate with the application. Applications can encourage the implementation of such features, though.

#### 4.4. Specifying URLs

In HTTP, the resources that clients interact with are identified with URLs [URL]. As [RFC8820] explains, parts of the URL are designed to be under the control of the owner (also known as the "authority") of that server, to give them the flexibility in deployment.

This means that in most cases, specifications for applications that use HTTP won't contain fixed application URLs or paths; while it is common practice for a specification of a single-deployment API to specify the path prefix `"/app/v1"` (for example), doing so in an IETF specification is inappropriate.

Therefore, the specification writer needs some mechanism to allow clients to discover an application's URLs. Additionally, they need to specify what URL scheme(s) the application should be used with, and whether to use a dedicated port, or reuse HTTP's port(s).

##### 4.4.1. Discovering an Application's URLs

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. Doing so assures that the deployment is as flexible as possible (potentially spanning multiple servers), allows evolution, and also gives the application the opportunity to tailor the 'discovery document' to the client.

There are a few common patterns for discovering that initial URL.

The most straightforward mechanism for URL discovery is to configure the client with (or otherwise convey to it) a full URL. This might be done in a configuration document, or through another discovery mechanism.

However, if the client only knows the server's hostname and the identity of the application, there needs to be some way to derive the initial URL from that information.

An application cannot define a fixed prefix for its URL paths; see [RFC8820]. Instead, a specification for such an application can use one of the following strategies:

- \* Register a Well-Known URI [WELL-KNOWN-URI] as an entry point for that application. This provides a fixed path on every potential server that will not collide with other applications.
- \* Enable the server authority to convey a URI Template [URI-TEMPLATE] or similar mechanism for generating a URL for an entry point. For example, this might be done in a configuration document or other artefact.

Once the discovery document is located, it can be fetched, cached for later reuse (if allowed by its metadata), and used to locate other resources that are relevant to the application, using full URIs or URL Templates.

In some cases, an application may not wish to use such a discovery document; for example, when communication is very brief, or when the latency concerns of doing so precludes the use of a discovery document. These situations can be addressed by placing all of the application's resources under a well-known location.

#### 4.4.2. Considering URI Schemes

Applications that use HTTP will typically employ the "http" and/or "https" URI 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 URI 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 URI schemes with Web browsers (e.g. `registerProtocolHandler()` in [HTML], 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 field), 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 [COOKIES], authentication [HTTP], caching [HTTP-CACHING], 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 URI 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 URI 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 (including those stemming from this distinguishability) should be documented in Security Considerations.

See [RFC7605] for further guidance.

#### 4.5. Using 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 [HTTP]), 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, [HTTP] now forbids this.

When authors believe that a new method is required, they are encouraged to engage with the HTTP community early (e.g., on the [ietf-http-wg@w3.org](mailto:ietf-http-wg@w3.org) mailing list), and document their proposal as a separate HTTP extension, rather than as part of an application's specification.

##### 4.5.1. GET

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

Queries can be performed with GET, 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 URI; in particular, if binary data forms part of the query terms, it needs to be encoded to conform to URI 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 [HTTP]).

In these cases, an application using HTTP might consider using POST to express queries in the request's content; 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 to consider allowing both methods.

Processing of GET requests should not change application state or have other side effects that might be significant to the client, since implementations can and do retry HTTP GET requests that fail, and some GET requests protected by TLS Early Data might be vulnerable to replay attacks (see [RFC8470]). Note that this does not include logging and similar functions; see [HTTP], Section 9.2.1.

Finally, note that while the generic HTTP syntax allows a GET request message to contain content, the purpose is to allow message parsers to be generic; as per [HTTP], Section 9.3.1, content on a GET is not recommended, has no meaning, and will be either ignored or rejected by generic HTTP software (such as intermediaries, caches, servers, and client libraries).

#### 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 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 [WELL-KNOWN-URI], or use an already existing one if appropriate (e.g., HostMeta [RFC6415]).

- \* For metadata about a specific resource, create a separate resource and link to it using a Link response header field or a link serialised into the response's content. See [WEB-LINKING]. Note that the Link header field 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. Using HTTP Status Codes

HTTP status codes convey semantics both for the benefit of generic HTTP components -- such as caches, intermediaries, and clients -- and applications themselves. However, applications can encounter a number of pitfalls in their use.

First, status codes are often generated by components other than the application itself. This can happen, for example, when network errors are encountered, a captive portal, proxy or Content Delivery Network is present, when a server is overloaded, or it thinks it is under attack. They can even be generated by generic client software when certain error conditions are encountered. As a result, if an application assigns specific semantics to one of these status codes, a client can be misled about its state, because the status code was generated by a generic component, not the application itself.

Furthermore, mapping application errors to individual HTTP status codes one-to-one often leads to a situation where the finite space of applicable HTTP status codes is exhausted. This, in turn, leads to a number of bad practices -- including minting new, application-specific status codes, or using existing status codes even though the link between their semantics and the application's is tenuous at best.

Instead, applications using HTTP should define their errors to use the most applicable status code, making generous use of the general status codes (200, 400 and 500) when in doubt. Importantly, they should not specify a one-to-one relationship between status codes and application errors, thereby avoiding the exhaustion issue outlined above.

To distinguish between multiple error conditions that are mapped to the same status code, and to avoid the misattribution issue outlined above, applications using HTTP should convey finer-grained error information in the response's message content and/or header fields. [PROBLEM-DETAILS] provides one way to do so.

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 won't be complete in the foreseeable future.

Applications using HTTP MUST NOT re-specify the semantics of HTTP status codes, even if it is only by copying their definition. It is NOT RECOMMENDED they require specific reason phrases to be used; the reason phrase has no function in HTTP, is not guaranteed to be preserved by implementations, and is not carried at all in the HTTP/2 HTTP2 message format.

Applications MUST only use registered HTTP status codes. As with methods, new HTTP status codes are rare, and required (by [HTTP]) to be registered with IETF Review. Similarly, HTTP status codes are generic; they are required (by [HTTP]) 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 (e.g., on the [ietf-http-wg@w3.org](mailto:ietf-http-wg@w3.org) mailing list), 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 Section 15.4 of [HTTP] direct the user agent to another resource to satisfy the request. The most common of these are 301, 302, 307 and 308, all of which use the Location response header field to indicate where the client should resend the request.

There are two ways that the members of 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 cannot. 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:

	Permanent	Temporary
Allows changing the request method from POST to GET	301	302
Does not allow changing the request method	308	307

Table 1

The 303 See Other status code can be used to inform the client that the result of an operation is available at a different location using GET.

As noted in [HTTP], 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.

Redirects can be cached (when appropriate cache directives are present), but beyond that they are not 'sticky' -- i.e., redirection of a URI will not result in the client assuming that similar URIs (e.g., with different query parameters) will also be redirected.

Applications using HTTP are encouraged to 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 ([HTTP], Section 11) and Cookie ([COOKIES]) header fields will change if the origin (and sometimes path) of the request changes. An application using HTTP should specify if any request header fields that it defines 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. Specifying HTTP Header Fields

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

- \* The field is useful to intermediaries (who often wish to avoid parsing message content), and/or

- \* The field is useful to generic HTTP software (e.g., clients, servers), and/or
- \* It is not possible to include their values in the message content (usually because a format does not allow it).

When the conditions above are not met, it is usually better to convey application-specific information in other places; e.g., the message content or the URL query string.

New header fields **MUST** be registered, as per Section 16.3 of [HTTP].

See Section 16.3.2 of [HTTP] for guidelines to consider when minting new header fields. [STRUCTURED-FIELDS] 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 field compression is used, there is an overhead) but appropriately specific. In particular, if a header field is specific to an application, an identifier for that application can form a prefix to the header field name, separated by a "-".

For example, if the "example" application needs to create three header fields, they might be called "example-foo", "example-bar" and "example-baz". Note that the primary motivation here is to avoid consuming more generic field 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 field has a special meaning in a certain context.

See Section 4.9 for the interaction between header fields and HTTP caching; in particular, request header fields 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 Content

Common syntactic conventions for message contents include JSON [JSON], XML [XML], and CBOR [RFC8949]. 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. Leveraging HTTP Caching

HTTP caching [HTTP-CACHING] 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.

Even when an application using HTTP isn't designed to take advantage of caching, it needs to consider how caches will handle its responses, to preserve correct behaviour when one is interposed (whether in the network, server, client, or intervening infrastructure).

##### 4.9.1. Freshness

Assigning even a short freshness lifetime ([HTTP-CACHING], 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.

The most common method for specifying freshness is the max-age response directive ([HTTP-CACHING], Section 5.2.2.1). The Expires header field ([HTTP-CACHING], Section 5.3) can also be used, but it is not necessary; all modern cache implementations support Cache-Control, and specifying freshness as a delta is usually more convenient and less error-prone.

It is not necessary to add the "public" response directive ([HTTP-CACHING], Section 5.2.2.9) to cache most responses; it is only necessary when it's desirable to store an authenticated response, or when the status code isn't understood by the cache and there isn't explicit freshness information available.

In some situations, responses without explicit cache freshness directives will be stored and served using a heuristic freshness lifetime; see [HTTP-CACHING], Section 4.2.2. As the heuristic is not

under control of the application, it is generally preferable to set an explicit freshness lifetime, or make the response explicitly uncacheable.

If caching of a response is not desired, the appropriate response directive is "Cache-Control: no-store". Other directives are not necessary, and no-store only need be sent in situations where the response might be cached; see [HTTP-CACHING], Section 3. Note that "Cache-Control: no-cache" allows a response to be stored, just not reused by a cache without validation; 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]

#### 4.9.2. Stale Responses

Authors should understand that stale responses (e.g., with "Cache-Control: max-age=0") can be reused by caches when disconnected from the origin server; this can be useful for handling network issues.

If doing so is not suitable for a given response, the origin should use "Cache-Control: must-revalidate". See Section 4.2.4 of [HTTP-CACHING], 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 Section 13 of [HTTP].

#### 4.9.3. Caching and Application Semantics

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

In particular, application authors need to consider how responses that are not freshly obtained from the origin server should be handled; if they have a concept like a validity period, this will need to be calculated considering the age of the response (see [HTTP-CACHING], Section 4.2.3).

One way to address this is to explicitly specify that responses need to be fresh upon use.

#### 4.9.4. Varying Content Based Upon the Request

If an application uses a request header field to change the response's header fields or content, 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 [HTTP-CACHING], Section 5.2.2.5) or send the Vary response header field ([HTTP], Section 12.5.5) 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.

#### 4.10. Handling Application State

Applications can use stateful cookies [COOKIES] 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.

#### 4.11. Making Multiple Requests

Clients often need to send multiple requests to perform a task.

In HTTP/1 [HTTP11], parallel requests are most often supported by opening multiple connections. Application performance can be impacted when too many simultaneous connections are used, because connections' congestion control will not be coordinated. Furthermore, it can be difficult for applications to decide when to issue and which connection to use for a given request, further impacting performance.

HTTP/2 [HTTP2] and HTTP/3 [HTTP3] offer multiplexing to applications, removing the need to use multiple connections. However, application performance can still be significantly affected by how the server chooses to prioritize responses. Depending on the application, it might be best for the server to determine the priority of responses, or for the client to hint its priorities to the server (see, e.g., [HTTP-PRIORITY]).

In all versions of HTTP, requests are made independently -- you can't rely on the relative order of two requests to guarantee processing order. This is because they might be sent over a multiplexed protocol by an intermediary, sent to different origin servers, or the server might even perform processing in a different order. If two requests need strict ordering, the only reliable way to assure the outcome is to issue the second request when the final response to the first has begun.

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.

#### 4.12. Client Authentication

Applications can use HTTP authentication Section 11 of [HTTP] to identify clients. As per [RFC7617], the Basic authentication scheme is not suitable for protecting sensitive or valuable information unless the channel is secure (e.g., using the "HTTPS" URI scheme). Likewise, [RFC7616] requires the Digest authentication scheme to be used over a secure channel.

With HTTPS, clients might also be authenticated using certificates [RFC8446], but note that such authentication is intrinsically scoped to the underlying transport connection. As a result, a client has no way of knowing whether the authenticated status was used in preparing the response (though "Vary: \*" and/or "Cache-Control: private" can provide a partial indication), and the only way to obtain a specifically unauthenticated response is to open a new connection.

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; see Section 8.3 of [RFC6454]), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

#### 4.13. 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 an attacker gains control of content returned from the application's resources (for example, part of the request is reflected in the response, or the response contains external information that the attacker can change), they can inject code into the browser and access data and capabilities as if they were the origin -- a technique known as a cross-site scripting (XSS) attack.

This is only a small sample of the kinds of issues that applications using HTTP must consider. Generally, the best approach is to actually consider the application 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 field, 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 (i.e., that which can execute scripts, such as HTML [HTML] and PDF), 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 field.



- \* Using the 'HttpOnly' flag on Cookies to assure that cookies are not exposed to browser scripting languages [COOKIES].
- \* 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.

#### 4.14. Maintaining Application Boundaries

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

For example, if Cookies [COOKIES] 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 [RFC8820] 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 [HTTP], 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].

#### 4.15. Using Server Push

HTTP/2 added the ability for servers to "push" request/response pairs to clients in [HTTP2], Section 8.4. 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 [HTTP2], Section 8.4, to avoid cross-origin push attacks.

#### 4.16. Allowing Versioning and Evolution

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

In HTTP, backwards-incompatible changes can be made using mechanisms such as:

- \* Using a distinct link relation type [WEB-LINKING] 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 content.

#### 5. IANA Considerations

This document has no requirements for IANA.

#### 6. Security Considerations

Applications using HTTP are subject to the security considerations of HTTP itself and any extensions used; [HTTP], [HTTP-CACHING], and [WEB-LINKING] are often relevant, amongst others.

Section 4.4.2 recommends support for 'https' URLs, and discourages the use of 'http' URLs, to provide authentication, integrity and confidentiality, as well as mitigate pervasive monitoring attacks. Many applications using HTTP perform authentication and authorization with bearer tokens (e.g., in session cookies). If the transport is unencrypted, an attacker that can eavesdrop upon or modify HTTP communications can often escalate their privilege to perform operations on resources.

Section 4.9.3 highlights the potential for mismatch between HTTP caching and application-specific storage of responses or information therein.

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

Section 4.13 highlights the implications of Web browsers' capabilities on applications that use HTTP.

Section 4.14 discusses the issues that arise when applications are deployed on the same origin as Web sites (and other applications).

Section 4.15 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 URI 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 code execution.

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 need 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 field conveys specific information about the implementation; the Accept-Language request header field 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 execute 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 to 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

- [HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-18, 18 August 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-18>>.
- [HTTP-CACHING] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-18, 18 August 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cache-18>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/rfc/rfc6454>>.
- [RFC6648] Saint-Andre, P., Crocker, D., and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols", BCP 178, RFC 6648, DOI 10.17487/RFC6648, June 2012, <<https://www.rfc-editor.org/rfc/rfc6648>>.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <<https://www.rfc-editor.org/rfc/rfc6838>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8820] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 8820, DOI 10.17487/RFC8820, June 2020, <<https://www.rfc-editor.org/rfc/rfc8820>>.
- [URL] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/rfc/rfc3986>>.
- [WEB-LINKING]  
Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <<https://www.rfc-editor.org/rfc/rfc8288>>.
- [WELL-KNOWN-URI]  
Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <<https://www.rfc-editor.org/rfc/rfc8615>>.

## 7.2. Informative References

- [COOKIES] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/rfc/rfc6265>>.
- [CSP] West, M., "Content Security Policy Level 3", World Wide Web Consortium WD WD-CSP3-20160913, 13 September 2016, <<https://www.w3.org/TR/2016/WD-CSP3-20160913>>.
- [FETCH] WHATWG, "Fetch - Living Standard", n.d., <<https://fetch.spec.whatwg.org>>.
- [HTML] WHATWG, "HTML - Living Standard", n.d., <<https://html.spec.whatwg.org>>.
- [HTTP-PRIORITY]  
Oku, K. and L. Pardue, "Extensible Prioritization Scheme for HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-priority-04, 11 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-priority-04>>.

- [HTTP11] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-18, 18 August 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-messaging-18>>.
- [HTTP2] Thomson, M. and C. Benfield, "Hypertext Transfer Protocol Version 2 (HTTP/2)", Work in Progress, Internet-Draft, draft-ietf-httpbis-http2bis-03, 12 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-http2bis-03>>.
- [HTTP3] Bishop, M., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-34, 2 February 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>>.
- [JSON] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/rfc/rfc8259>>.
- [PROBLEM-DETAILS] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <<https://www.rfc-editor.org/rfc/rfc7807>>.
- [REFERRER-POLICY] Eisinger, J. and E. Stark, "Referrer Policy", World Wide Web Consortium CR CR-referrer-policy-20170126, 26 January 2017, <<https://www.w3.org/TR/2017/CR-referrer-policy-20170126>>.
- [RFC3205] Moore, K., "On the use of HTTP as a Substrate", BCP 56, RFC 3205, DOI 10.17487/RFC3205, February 2002, <<https://www.rfc-editor.org/rfc/rfc3205>>.
- [RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <<https://www.rfc-editor.org/rfc/rfc4791>>.
- [RFC4918] Dusseault, L., Ed., "HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)", RFC 4918, DOI 10.17487/RFC4918, June 2007, <<https://www.rfc-editor.org/rfc/rfc4918>>.

- [RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale Content", RFC 5861, DOI 10.17487/RFC5861, May 2010, <<https://www.rfc-editor.org/rfc/rfc5861>>.
- [RFC6415] Hammer-Lahav, E., Ed. and B. Cook, "Web Host Metadata", RFC 6415, DOI 10.17487/RFC6415, October 2011, <<https://www.rfc-editor.org/rfc/rfc6415>>.
- [RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <<https://www.rfc-editor.org/rfc/rfc6797>>.
- [RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<https://www.rfc-editor.org/rfc/rfc7258>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/rfc/rfc7301>>.
- [RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <<https://www.rfc-editor.org/rfc/rfc7595>>.
- [RFC7605] Touch, J., "Recommendations on Using Assigned Transport Port Numbers", BCP 165, RFC 7605, DOI 10.17487/RFC7605, August 2015, <<https://www.rfc-editor.org/rfc/rfc7605>>.
- [RFC7616] Shekh-Yusef, R., Ed., Ahrens, D., and S. Bremer, "HTTP Digest Access Authentication", RFC 7616, DOI 10.17487/RFC7616, September 2015, <<https://www.rfc-editor.org/rfc/rfc7616>>.
- [RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <<https://www.rfc-editor.org/rfc/rfc7617>>.
- [RFC8297] Oku, K., "An HTTP Status Code for Indicating Hints", RFC 8297, DOI 10.17487/RFC8297, December 2017, <<https://www.rfc-editor.org/rfc/rfc8297>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.



- [RFC8470] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/rfc/rfc8470>>.
- [RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <<https://www.rfc-editor.org/rfc/rfc8949>>.
- [SECCTXT] West, M., "Secure Contexts", World Wide Web Consortium CR CR-secure-contexts-20160915, 15 September 2016, <<https://www.w3.org/TR/2016/CR-secure-contexts-20160915>>.
- [STRUCTURED-FIELDS]  
Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <<https://www.rfc-editor.org/rfc/rfc8941>>.
- [URI-TEMPLATE]  
Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <<https://www.rfc-editor.org/rfc/rfc6570>>.
- [XML] Bray, T., Paoli, J., Sperberg-McQueen, M., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, 26 November 2008, <<https://www.w3.org/TR/2008/REC-xml-20081126>>.

#### 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  
Pahran  
Australia

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

HTTP  
Internet-Draft  
Intended status: Standards Track  
Expires: 18 February 2022

M. Nottingham  
Fastly  
17 August 2021

The Cache-Status HTTP Response Header Field  
draft-ietf-httpbis-cache-header-10

Abstract

To aid debugging, HTTP caches often append header fields to a response explaining how they handled the request in an ad hoc manner. This specification defines a standard mechanism to do so that is aligned with HTTP's 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](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> (<https://lists.w3.org/Archives/Public/ietf-http-wg/>).

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

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 18 February 2022.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	2
1.1. Notational Conventions . . . . .	3
2. The Cache-Status HTTP Response Header Field . . . . .	3
2.1. The hit parameter . . . . .	4
2.2. The fwd parameter . . . . .	4
2.3. The fwd-status parameter . . . . .	5
2.4. The ttl parameter . . . . .	6
2.5. The stored parameter . . . . .	6
2.6. The collapsed parameter . . . . .	6
2.7. The key parameter . . . . .	6
2.8. The detail parameter . . . . .	6
3. Examples . . . . .	7
4. Defining New Cache-Status Parameters . . . . .	8
5. IANA Considerations . . . . .	8
6. Security Considerations . . . . .	9
7. References . . . . .	9
7.1. Normative References . . . . .	9
7.2. Informative References . . . . .	10
Author's Address . . . . .	10

## 1. Introduction

To aid debugging (both by humans and automated tools), HTTP caches often append header fields to a response explaining how they handled the request. Unfortunately, the semantics of these headers are often unclear, and both the semantics and syntax used vary between implementations.

This specification defines a new HTTP response header field, "Cache-Status" for this purpose, with standardized syntax and semantics.

### 1.1. Notational Conventions

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

This document uses ABNF as defined in [RFC5234], with rules prefixed with "sf-" and the "key" rule as defined in [STRUCTURED-FIELDS]. It uses terminology from [HTTP] and [HTTP-CACHING].

## 2. The Cache-Status HTTP Response Header Field

The Cache-Status HTTP response header field indicates how caches have handled that response and its corresponding request. The syntax of this header field conforms to [STRUCTURED-FIELDS].

Its value is a List ([STRUCTURED-FIELDS], Section 3.1):

Cache-Status = sf-list

Each member of the 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 (possibly including the user agent's cache itself, if it appends a value).

Caches determine when it is appropriate to add the Cache-Status header field to a response. Some might add it to all responses, whereas others might only do so when specifically configured to, or when the request contains a header field that activates a debugging mode. See Section 6 for related security considerations.

An intermediary SHOULD NOT append a Cache-Status member to responses that it generates locally, even if that intermediary contains a cache, unless the generated response is based upon a stored response (e.g., 304 Not Modified and 206 Partial Content are both based upon a stored response). For example, a proxy generating a 400 response due to a malformed request will not add a Cache-Status value, because that response was generated by the proxy, not the origin server.

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

Each list member identifies the cache that inserted it and this identifier MUST be a String or Token. Depending on the deployment, this might be a product or service name (e.g., ExampleCache or "Example CDN"), a hostname ("cache-3.example.com"), an IP address, or a generated string.

Each member of the list can have parameters that describe that cache's handling of the request. While these parameters are OPTIONAL, caches are encouraged to provide as much information as possible.

This specification defines the following parameters:

hit	= sf-boolean
fwd	= sf-token
fwd-status	= sf-integer
ttl	= sf-integer
stored	= sf-boolean
collapsed	= sf-boolean
key	= sf-string
detail	= sf-token / sf-string

### 2.1. The hit parameter

"hit", when true, indicates that the request was satisfied by the cache; i.e., it was not forwarded, and the response was obtained from the cache.

A response that was originally produced by the origin but was modified by the cache (for example, a 304 or 206 status code) is still considered a hit, as long as it did not go forward (e.g., for validation).

A response that was in cache but not able to be used without going forward (e.g., because it was stale, or partial) is not considered a hit. Note that a stale response that is used without going forward (e.g., because the origin server is not available) can be considered a hit.

"hit" and "fwd" are exclusive; only one of them should appear on each list member.

### 2.2. The fwd parameter

"fwd" indicates that the request went forward towards the origin, and why.

The following parameter values are defined to explain why the request went forward, from most specific to least:

- \* bypass - The cache was configured to not handle this request
- \* method - The request method's semantics require the request to be forwarded
- \* uri-miss - The cache did not contain any responses that matched the request URI
- \* vary-miss - The cache contained a response that matched the request URI, but could not select a response based upon this request's headers and stored Vary headers.
- \* miss - The cache did not contain any responses that could be used to satisfy this request (to be used when an implementation cannot distinguish between uri-miss and vary-miss)
- \* request - The cache was able to select a fresh response for the request, but the request's semantics (e.g., Cache-Control request directives) did not allow its use
- \* stale - The cache was able to select a response for the request, but it was stale
- \* partial - The cache was able to select a partial response for the request, but it did not contain all of the requested ranges (or the request was for the complete response)

The most specific reason that the cache is aware of SHOULD be used, to the extent that it is possible to implement. See also [HTTP-CACHING], Section 4.

### 2.3. The fwd-status parameter

"fwd-status" indicates what status code (see [HTTP], Section 15) the next hop server returned in response to the forwarded request. Only meaningful when "fwd" is present; if "fwd-status" is not present but "fwd" is, it defaults to the status code sent in the response.

This parameter is useful to distinguish cases when the next hop server sends a 304 Not Modified response to a conditional request, or a 206 Partial Response because of a range request.

#### 2.4. The ttl parameter

"ttl" indicates the response's remaining freshness lifetime (see [HTTP-CACHING], Section 4.2.1) as calculated by the cache, as an integer number of seconds, measured as closely as possible to when the response header section is sent by the cache. This includes freshness assigned by the cache; e.g., through heuristics (see [HTTP-CACHING], Section 4.2.2), local configuration, or other factors. May be negative, to indicate staleness.

#### 2.5. The stored parameter

"stored" indicates whether the cache stored the response (see [HTTP-CACHING], Section 3); a true value indicates that it did. Only meaningful when fwd is present.

#### 2.6. The collapsed parameter

"collapsed" indicates whether this request was collapsed together with one or more other forward requests (see [HTTP-CACHING], Section 4); if true, the response was successfully reused; if not, a new request had to be made. If not present, the request was not collapsed with others. Only meaningful when fwd is present.

#### 2.7. The key parameter

"key" conveys a representation of the cache key (see [HTTP-CACHING], Section 2) used for the response. Note that this may be implementation-specific.

#### 2.8. The detail parameter

"detail" allows implementations to convey additional information not captured in other parameters; for example, implementation-specific states, or other caching-related metrics.

For example:

```
Cache-Status: ExampleCache; hit; detail=MEMORY
```

The semantics of a detail parameter are always specific to the cache that sent it; even if a member of details from another cache shares the same name, it might not mean the same thing.

This parameter is intentionally limited. If an implementation's developer or operator needs to convey additional information in an interoperable fashion, they are encouraged to register extension parameters (see Section 4) or define another header field.

### 3. Examples

The most minimal cache hit:

```
Cache-Status: ExampleCache; hit
```

... but a polite cache will give some more information, e.g.:

```
Cache-Status: ExampleCache; hit; ttl=376
```

A stale hit just has negative freshness:

```
Cache-Status: ExampleCache; hit; ttl=-412
```

Whereas a complete miss is:

```
Cache-Status: ExampleCache; fwd=uri-miss
```

A miss that successfully validated on the back-end server:

```
Cache-Status: ExampleCache; fwd=stale; fwd-status=304
```

A miss that was collapsed with another request:

```
Cache-Status: ExampleCache; fwd=uri-miss; collapsed
```

A miss that the cache attempted to collapse, but couldn't:

```
Cache-Status: ExampleCache; fwd=uri-miss; collapsed=?0
```

Going through two separate layers of caching, where the cache closest to the origin responded to an earlier request with a stored response, and a second cache stored that response and later reused it to satisfy the current request:

```
Cache-Status: OriginCache; hit; ttl=1100,  
             "CDN Company Here"; hit; ttl=545
```

Going through a three-layer caching system, where the closest to the origin is a reverse proxy (where the response was served from cache), the next is a forward proxy interposed by the network (where the request was forwarded because there wasn't any response cached with its URI, the request was collapsed with others, and the resulting response was stored), and the closest to the user is a browser cache (where there wasn't any response cached with the request's URI):



```
Cache-Status: ReverseProxyCache; hit
Cache-Status: ForwardProxyCache; fwd=uri-miss; collapsed; stored
Cache-Status: BrowserCache; fwd=uri-miss
```

#### 4. Defining New Cache-Status Parameters

New Cache-Status Parameters can be defined by registering them in the HTTP Cache-Status Parameters 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:

- \* Community feedback
- \* If the value is sufficiently well-defined
- \* Generic parameters are preferred over vendor-specific, application-specific, or deployment-specific values. If a generic value cannot be agreed upon in the community, the parameter's name should be correspondingly specific (e.g., with a prefix that identifies the vendor, application or deployment).

Registration requests should use the following template:

- \* Name: [a name for the Cache-Status Parameter that matches the 'key' ABNF rule]
- \* Description: [a description of the parameter semantics and value]
- \* Reference: [to a specification defining this parameter, if available]

See the registry at <https://iana.org/assignments/http-cache-status> (<https://iana.org/assignments/http-cache-status>) for details on where to send registration requests.

#### 5. IANA Considerations

Upon publication, please create the HTTP Cache-Status Parameters registry at <https://iana.org/assignments/http-cache-status> (<https://iana.org/assignments/http-cache-status>) and populate it with the types defined in Section 2; see Section 4 for its associated procedures.

Also, please create the following entry in the Hypertext Transfer Protocol (HTTP) Field Name Registry defined in [HTTP], Section 18.4:

- \* Field name: Cache-Status
- \* Status: permanent
- \* Specification document: [this document]
- \* Comments:

## 6. Security Considerations

Attackers can use the information in Cache-Status to probe the behaviour of the cache (and other components), and infer the activity of those using the cache. The Cache-Status header field may not create these risks on its own, but can assist attackers in exploiting them.

For example, knowing if a cache has stored a response can help an attacker execute a timing attack on sensitive data.

Additionally, exposing the cache key can help an attacker understand modifications to the cache key, which may assist cache poisoning attacks. See [ENTANGLE] for details.

The underlying risks can be mitigated with a variety of techniques (e.g., use of encryption and authentication; avoiding the inclusion of attacker-controlled data in the cache key), depending on their exact nature. Note that merely obfuscating the key does not mitigate this risk.

To avoid assisting such attacks, the Cache-Status header field can be omitted, only sent when the client is authorized to receive it, or only send sensitive information (e.g., the key parameter) when the client is authorized.

## 7. References

### 7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.

- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/rfc/rfc8126>>.
- [STRUCTURED-FIELDS] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <<https://www.rfc-editor.org/rfc/rfc8941>>.
- [HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-17, 25 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-17>>.
- [HTTP-CACHING] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-17, 25 July 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cache-17>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/rfc/rfc5234>>.

## 7.2. Informative References

- [ENTANGLE] Kettle, J., "Web Cache Entanglement: Novel Pathways to Poisoning", 2020, <<https://i.blackhat.com/USA-20/Wednesday/us-20-Kettle-Web-Cache-Entanglement-Novel-Pathways-To-Poisoning-wp.pdf>>.

### Author's Address

Mark Nottingham  
Fastly  
Pahran VIC  
Australia

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

HTTP Working Group  
Internet-Draft  
Intended status: Experimental  
Expires: January 4, 2021

I. Grigorik  
Y. Weiss  
Google  
July 3, 2020

HTTP Client Hints  
draft-ietf-httpbis-client-hints-15

Abstract

HTTP defines proactive content negotiation to allow servers to select the appropriate response for a given request, based upon the user agent's characteristics, as expressed in request headers. In practice, user agents 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 an Accept-CH response header 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."

Note to Readers

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

Working Group information can be found at <http://httpwg.github.io/> [2]; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/client-hints> [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 January 4, 2021.

## Copyright Notice

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

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

## Table of Contents

1. Introduction . . . . .	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
3. Advertising Server Support . . . . .	5
3.1. The Accept-CH Response Header Field . . . . .	5
3.2. Interaction with Caches . . . . .	6
4. Security Considerations . . . . .	7
4.1. Information Exposure . . . . .	7
4.2. Deployment and Security Risks . . . . .	9
4.3. Abuse Detection . . . . .	9
5. Cost of Sending Hints . . . . .	9
6. IANA Considerations . . . . .	10
6.1. Accept-CH . . . . .	10
7. References . . . . .	10
7.1. Normative References . . . . .	10
7.2. Informative References . . . . .	11
7.3. URIs . . . . .	11
Appendix A. Changes . . . . .	11
A.1. Since -00 . . . . .	11
A.2. Since -01 . . . . .	12
A.3. Since -02 . . . . .	12
A.4. Since -03 . . . . .	12
A.5. Since -04 . . . . .	12
A.6. Since -05 . . . . .	12
A.7. Since -06 . . . . .	12
A.8. Since -07 . . . . .	12
A.9. Since -08 . . . . .	13

A.10. Since -09 . . . . .	13
A.11. Since -10 . . . . .	13
A.12. Since -11 . . . . .	13
A.13. Since -12 . . . . .	13
A.14. Since -13 . . . . .	13
A.15. Since -14 . . . . .	13
Acknowledgements . . . . .	13
Authors' Addresses . . . . .	13

## 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 user agent preferences. Historically, applications that wanted the server to optimize content delivery and user experience based on such capabilities had to rely on passive identification (e.g., by matching the User-Agent header field (Section 5.5.3 of [RFC7231]) against an established database of user agent signatures), use HTTP cookies [RFC6265] and URL parameters, or use some combination of these and similar mechanisms to enable ad hoc content negotiation.

Such techniques are expensive to set up and maintain, and are not portable across both applications and servers. They also make it hard for both user agent and server to understand which data are required and is in use during the negotiation:

- o User agent detection cannot reliably identify all static variables, cannot infer dynamic user agent preferences, requires an external device database, is not cache friendly, and is reliant on a passive fingerprinting surface.
- o Cookie-based approaches are not portable across applications and servers, impose additional client-side latency by requiring JavaScript execution, and are not cache friendly.
- o URL parameters, similar to cookie-based approaches, suffer from lack of portability, and are hard to deploy due to a requirement to encode content negotiation data inside of the URL of each resource.

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 based on those request headers (or on other, implicit characteristics).

However, traditional proactive content negotiation techniques often mean that user agents 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.

This document defines Client Hints, a framework that enables servers to opt-in to specific proactive content negotiation features, adapting their content accordingly, as well as guidelines for content negotiation mechanisms that use the framework. This document also defines a new response header, `Accept-CH`, that allows an origin server to explicitly ask that user agents send these headers in requests.

Client Hints mitigate performance concerns by assuring that user agents will only send the request headers when they're actually going to be used, and 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, turning passive fingerprinting vectors into active ones.

The document does not define specific usages of Client Hints. Such usages need to be defined in their respective specifications.

One example of such usage is the User Agent Client Hints [`UA-CH`].

### 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].

## 2. Client Hint Request Header Fields

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

### 2.1. Sending Client Hints

User agents choose what Client Hints to send in a request based on their default settings, user configuration, and server preferences expressed in "`Accept-CH`". The user agent and server can use an opt-

in mechanism outlined below to negotiate which header fields need to be sent to allow for efficient content adaption, and optionally use additional mechanisms (e.g., as outlined in [CLIENT-HINTS-INFRASTRUCTURE]) to negotiate delegation policies that control access of third parties to those same header fields. User agents SHOULD require an opt-in to send any hints that are not listed in the low-entropy hint table at [CLIENT-HINTS-INFRASTRUCTURE].

Implementers need to be aware of the fingerprinting implications when implementing support for Client Hints, and follow the considerations outlined in the Security Considerations (Section 4) 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.

Servers MUST ignore hints they do not understand nor support. There is no mechanism for servers to indicate to user agents that hints were ignored.

Furthermore, the server can generate additional response header fields (as specified by the hint or hints in use) that convey related values to aid client processing.

## 3. Advertising Server Support

Servers can advertise support for Client Hints using the mechanism described below.

### 3.1. The Accept-CH Response Header Field

The Accept-CH response header field indicates server support for the hints indicated in its value. Servers wishing to receive user agent information through Client Hints SHOULD add Accept-CH response header to their responses as early as possible.

Accept-CH is a Structured Header [I-D.ietf-httpbis-header-structure]. Its value MUST be an sf-list (Section 3.1 of [I-D.ietf-httpbis-header-structure]) whose members are tokens (Section 3.3.4 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:



Accept-CH = sf-list

For example:

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

When a user agent receives an HTTP response containing "Accept-CH", that indicates that the origin opts-in to receive the indicated request header fields for subsequent same-origin requests. The opt-in MUST be ignored if delivered over non-secure transport (using a scheme different from HTTPS). It SHOULD be persisted and bound to the origin to enable delivery of Client Hints on subsequent requests to the server's origin, for the duration of the user's session (as defined by the user agent). An opt-in overrides previous persisted opt-in values and SHOULD be persisted in its stead.

Based on the Accept-CH example above, which is received in response to a user agent navigating to "https://site.example", and delivered over a secure transport, persisted Accept-CH preferences will be bound to "https://site.example". It will then use it for navigations to e.g., "https://site.example/foobar.html", but not to e.g., "https://foobar.site.example/". It will similarly use the preference for any same-origin resource requests (e.g., to "https://site.example/image.jpg") initiated by the page constructed from the navigation's response, but not to cross-origin resource requests (e.g., "https://thirdparty.example/resource.js"). This preference will not extend to resource requests initiated to "https://site.example" from other origins (e.g., from navigations to "https://other.example/").

### 3.2. Interaction with Caches

When selecting a 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

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

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

The above example indicates that the cache key needs to include the Sec-CH-Example and Sec-CH-Example-2 header fields.

## 4. Security Considerations

### 4.1. Information Exposure

Request header fields used in features relying on this document expose information about the user's environment to enable privacy-preserving proactive content negotiation, and avoid exposing passive fingerprinting vectors. However, implementers need to bear in mind that in the worst case, uncontrolled and unmonitored active fingerprinting is not better than passive fingerprinting. In order to provide user privacy benefits, user agents need to apply further policies that prevent abuse of the information exposed by features using Client Hints.

The information exposed by features might reveal new information about the user and implementers ought to consider the following considerations, recommendations, and best practices.

The underlying assumption is that exposing information about the user as a request header is equivalent (from a security perspective) to exposing this information by other means. (For example, if the request's origin can access that information using JavaScript APIs, and transmit it to its servers).

Because Client Hints is an explicit opt-in mechanism, that means that servers that want access to information about the user's environment need to actively ask for it, enabling clients and privacy researchers to keep track of which origins collect that data, and potentially act upon it. The header-based opt-in means that removal of passive fingerprinting vectors is possible, such as the User-Agent string (enabling active access to that information through User-Agent Client Hints ([UA-CH]) or otherwise expose information already available through script (e.g., the Save-Data Client Hint [4]), without increasing the passive fingerprinting surface. User agents supporting Client Hints features which send certain information to opted-in servers SHOULD avoid sending the equivalent information passively.

Therefore, features relying on this document to define Client Hint headers MUST NOT provide new information that is otherwise not made available to the application by the user agent, such as existing request headers, HTML, CSS, or JavaScript.

Such features need to take into account the following aspects of the information exposed:

- o Entropy - Exposing highly granular data can be used to help identify users across multiple requests to different origins.

Reducing the set of header field values that can be expressed, or restricting them to an enumerated range where the advertised value is close to 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.

- o Sensitivity - The feature SHOULD NOT expose user-sensitive information. To that end, information available to the application, but gated behind specific user actions (e.g., a permission prompt or user activation) SHOULD NOT be exposed as a Client Hint.
- o Change over time - The feature SHOULD NOT expose user information that changes over time, unless the state change itself is also exposed (e.g., through JavaScript callbacks).

Different features will be positioned in different points in the space between low-entropy, non-sensitive and static information (e.g., user agent information), and high-entropy, sensitive and dynamic information (e.g., geolocation). User agents need to consider the value provided by a particular feature vs these considerations, and may wish to have different policies regarding that tradeoff on a per-feature or other fine-grained basis.

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

- o 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.
- o Implementers considering providing user choice mechanisms that allow users to balance privacy concerns against bandwidth limitations need to also consider that explaining to users the privacy implications involved, such as the risks of passive fingerprinting, may be challenging or even impractical.
- o 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.

User agents MUST clear persisted opt-in preferences when any one of site data, browsing history, browsing cache, cookies, or similar, are cleared.

#### 4.2. Deployment and Security Risks

Deployment of new request headers requires several considerations:

- o Potential conflicts due to existing use of header field name
- o Properties of the data communicated in header field value

Authors of new Client Hints are advised to carefully consider whether they need to be able to be added by client-side content (e.g., scripts), or whether they need to be exclusively set by the user agent. In the latter case, the Sec- prefix on the header field name has the effect of preventing scripts and other application content from setting them in user agents. Using the "Sec-" prefix signals to servers that the user agent - and not application content - generated the values. See [FETCH] for more information.

By convention, request headers that are Client Hints are encouraged to use a CH- prefix, to make them easier to identify as using this framework; for example, CH-Foo or, with a "Sec-" prefix, Sec-CH-Foo. Doing so makes them easier to identify programmatically (e.g., for stripping unrecognised hints from requests by privacy filters).

A Client Hints request header negotiated using the Accept-CH opt-in mechanism MUST have a field name that matches sf-token (Section 3.3.4 of [I-D.ietf-httpbis-header-structure]).

#### 4.3. Abuse Detection

A user agent that tracks access to active fingerprinting information SHOULD consider emission of Client Hints headers similarly to the way it would consider access to the equivalent API.

Research into abuse of Client Hints might look at how HTTP responses to requests that contain Client Hints differ from those with different values, and from those without. This might be used to reveal which Client Hints are in use, allowing researchers to further analyze that use.

#### 5. Cost of Sending Hints

Sending Client Hints to the server incurs an increase in request byte size. Some of this increase can be mitigated by HTTP header compression schemes, but each new hint sent will still lead to some increased bandwidth usage. Servers SHOULD take that into account when opting in to receive Client Hints, and SHOULD NOT opt-in to receive hints unless they are to be used for content adaptation purposes.

Due to request byte size increase, features relying on this document to define Client Hints MAY consider restricting sending those hints to certain request destinations [FETCH], where they are more likely to be useful.

## 6. IANA Considerations

Features relying on this document are expected to register added request header fields in the Permanent Message Header Fields registry ([RFC3864]).

This document defines the "Accept-CH" HTTP response header field, and registers it in the same registry.

### 6.1. Accept-CH

- o Header field name: Accept-CH
- o Applicable protocol: HTTP
- o Status: experimental
- o Author/Change controller: IETF
- o Specification document(s): Section 3.1 of this document
- o Related information: for Client Hints

## 7. References

### 7.1. Normative References

- [CLIENT-HINTS-INFRASTRUCTURE]  
Weiss, Y., "Client Hints Infrastructure", n.d.,  
<<https://wicg.github.io/client-hints-infrastructure/>>.
- [I-D.ietf-httpbis-header-structure]  
Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", draft-ietf-httpbis-header-structure-19 (work in progress), June 2020.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <<https://www.rfc-editor.org/info/rfc3864>>.

- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <<https://www.rfc-editor.org/info/rfc7234>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

## 7.2. Informative References

- [FETCH] van Kesteren, A., "Fetch", n.d., <<https://fetch.spec.whatwg.org/>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [UA-CH] West, M. and Y. Weiss, "User Agent Client Hints", n.d., <<https://wicg.github.io/ua-client-hints/>>.

## 7.3. URIs

- [1] <https://lists.w3.org/Archives/Public/ietf-http-wg/>
- [2] <http://httpwg.github.io/>
- [3] <https://github.com/httpwg/http-extensions/labels/client-hints>
- [4] <https://wicg.github.io/savedata/#save-data-request-header-field>

## Appendix A. Changes

### A.1. Since -00

- o Issue 168 (make Save-Data extensible) updated ABNF.
- o Issue 163 (CH review feedback) editorial feedback from httpwg list.

- o Issue 153 (NetInfo API citation) added normative reference.
- A.2. Since -01
- o Issue 200: Moved Key reference to informative.
  - o Issue 215: Extended passive fingerprinting and mitigation considerations.
  - o Changed document status to experimental.
- A.3. Since -02
- o Issue 239: Updated reference to CR-css-values-3
  - o Issue 240: Updated reference for Network Information API
  - o Issue 241: Consistency in IANA considerations
  - o Issue 250: Clarified Accept-CH
- A.4. Since -03
- o Issue 284: Extended guidance for Accept-CH
  - o Issue 308: Editorial cleanup
  - o Issue 306: Define Accept-CH-Lifetime
- A.5. Since -04
- o Issue 361: Removed Downlink
  - o Issue 361: Moved Key to appendix, plus other editorial feedback
- A.6. Since -05
- o Issue 372: Scoped CH opt-in and delivery to secure transports
  - o Issue 373: Bind CH opt-in to origin
- A.7. Since -06
- o Issue 524: Save-Data is now defined by NetInfo spec, dropping
  - o PR 775: Removed specific features to be defined in other specifications
- A.8. Since -07
- o Issue 761: Clarified that the defined headers are response headers.
  - o Issue 730: Replaced Key reference with Variants.
  - o Issue 700: Replaced ABNF with structured headers.
  - o PR 878: Removed Accept-CH-Lifetime based on feedback at IETF 105

## A.9. Since -08

- o PR 985: Describe the bytesize cost of hints.
- o PR 776: Add Sec- and CH- prefix considerations.
- o PR 1001: Clear CH persistence when cookies are cleared.

## A.10. Since -09

- o PR 1064: Fix merge issues with "cost of sending hints".

## A.11. Since -10

- o PR 1072: LC feedback from Julian Reschke.
- o PR 1080: Improve list style.
- o PR 1082: Remove section mentioning Variants.
- o PR 1097: Editorial feedback from mnot.
- o PR 1131: Remove unused references.
- o PR 1132: Remove nested list.

## A.12. Since -11

- o PR 1134: Re-insert back section.

## A.13. Since -12

- o PR 1160: AD review.

## A.14. Since -13

- o PR 1171: Genart review.

## A.15. Since -14

- o PR 1220: AD review.

## Acknowledgements

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

## Authors' Addresses



Ilya Grigorik  
Google

Email: [ilya@igvita.com](mailto:ilya@igvita.com)  
URI: <https://www.igvita.com/>

Yoav Weiss  
Google

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

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

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

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

Abstract

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

Note to Readers

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

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

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

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on December 5, 2020.

#### Copyright Notice

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

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

#### Table of Contents

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

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

## 1. Introduction

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

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

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

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

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

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

Section 2 describes how to specify a Structured Field.

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

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

### 1.1. Intentionally Strict Processing

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

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

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

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

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

## 1.2. Notational Conventions

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

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

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

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

## 2. Defining New Structured Fields

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

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

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

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

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

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

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

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

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

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

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

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

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

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



--8<--

#### 42. Foo-Example Header

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

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

Foo-Example = sf-integer

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

The following parameters are defined:

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

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

For example:

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

### 3. Structured Data Types

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

In summary:

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

### 3.1. Lists

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

The ABNF for Lists in HTTP fields is:

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

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

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

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

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

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

and

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

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

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

#### 3.1.1. Inner Lists

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

The ABNF for Inner Lists is:

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

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

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

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

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

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

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

### 3.1.2. Parameters

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

The ABNF for Parameters is:

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

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

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

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

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

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

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

### 3.2. Dictionaries

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

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

The ABNF for Dictionaries is:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example-Hdr: foo=1, bar=2

and

Example-Hdr: foo=1

Example-Hdr: bar=2

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

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

### 3.3. Items

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

The ABNF for Items is:

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

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

Example-IntItemHeader: 5

or with Parameters:

Example-IntItem: 5; foo=bar

### 3.3.1. Integers

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

The ABNF for Integers is:

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

For example:

Example-Integer: 42

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

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

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

### 3.3.2. Decimals

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

The ABNF for decimals is:

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

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

Example-Decimal: 4.5

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

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

### 3.3.3. Strings

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

The ABNF for Strings is:

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

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

Example-String: "hello world"

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

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

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

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

### 3.3.4. Tokens

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

The ABNF for Tokens is:

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

For example:

Example-Token: fool23/456

Parsers MUST support Tokens with at least 512 characters.

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

### 3.3.5. Byte Sequences

Byte Sequences can be conveyed in Structured Fields.

The ABNF for a Byte Sequence is:

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

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

Example-Binary: :cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==:

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

### 3.3.6. Booleans

Boolean values can be conveyed in Structured Fields.

The ABNF for a Boolean is:

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

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



Example-Bool: ?1

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

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

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

##### 4.1. Serializing Structured Fields

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

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

###### 4.1.1. Serializing a List

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

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

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

#### 4.1.1.1. Serializing an Inner List

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

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

#### 4.1.1.2. Serializing Parameters

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

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

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

#### 4.1.1.3. Serializing a Key

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

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

#### 4.1.2. Serializing a Dictionary

Given an ordered Dictionary as `input_dictionary` (each member having a `member_name` and a tuple value of (`member_value`, `parameters`)), return an ASCII string suitable for use in a HTTP field value.

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

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

#### 4.1.3. Serializing an Item

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

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

##### 4.1.3.1. Serializing a Bare Item

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

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

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

#### 4.1.4. Serializing an Integer

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

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

#### 4.1.5. Serializing a Decimal

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

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

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

#### 4.1.6. Serializing a String

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

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

#### 4.1.7. Serializing a Token

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

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

#### 4.1.8. Serializing a Byte Sequence

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

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

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

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

#### 4.1.9. Serializing a Boolean

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

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

#### 4.2. Parsing Structured Fields

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

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

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

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



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

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

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

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

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

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

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

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

#### 4.2.1. Parsing a List

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

1. Let `members` be an empty array.

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

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

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

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

#### 4.2.1.2. Parsing an Inner List

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

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

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

#### 4.2.2. Parsing a Dictionary

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

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

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

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

#### 4.2.3. Parsing an Item

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

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

##### 4.2.3.1. Parsing a Bare Item

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

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

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

#### 4.2.3.2. Parsing Parameters

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

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

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

#### 4.2.3.3. Parsing a Key

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

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

#### 4.2.4. Parsing an Integer or Decimal

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

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

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

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

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

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

#### 4.2.6. Parsing a Token

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

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



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

#### 4.2.7. Parsing a Byte Sequence

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

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

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

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

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

#### 4.2.8. Parsing a Boolean

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

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

#### 5. IANA Considerations

This document has no actions for IANA.

#### 6. Security Considerations

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

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

#### 7. References

##### 7.1. Normative References

- [RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <<https://www.rfc-editor.org/info/rfc20>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

## 7.2. Informative References

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

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

### 7.3. URIs

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

## Appendix A. Frequently Asked Questions

### A.1. Why not JSON?

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

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

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

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

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

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

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

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

## Appendix B. Implementation Notes

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

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

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

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

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

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

## Appendix C. Changes

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

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

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

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

- o Editorial improvements.

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

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

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

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

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

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



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

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

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

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

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

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

#### Acknowledgements

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

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

#### Authors' Addresses

Mark Nottingham  
Fastly

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

Poul-Henning Kamp  
The Varnish Cache Project

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

HTTP  
Internet-Draft  
Intended status: Standards Track  
Expires: November 15, 2020

M. Bishop  
Akamai  
N. Sullivan  
Cloudflare  
M. Thomson  
Mozilla  
May 14, 2020

Secondary Certificate Authentication in HTTP/2  
draft-ietf-httpbis-http2-secondary-certs-06

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](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> [1].

Working Group information can be found at <http://httpwg.github.io/> [2]; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/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 November 15, 2020.

## Copyright Notice

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

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

## Table of Contents

1. Introduction . . . . .	3
1.1. Server Certificate Authentication . . . . .	4
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 . . . . .	9
2.2. Making Certificates or Requests Available . . . . .	10
2.3. Requiring Certificate Authentication . . . . .	11
2.3.1. Requiring Additional Server Certificates . . . . .	11
2.3.2. Requiring Additional Client Certificates . . . . .	12
3. Certificates Frames for HTTP/2 . . . . .	13
3.1. The CERTIFICATE_NEEDED Frame . . . . .	13
3.2. The USE_CERTIFICATE Frame . . . . .	15
3.3. The CERTIFICATE_REQUEST Frame . . . . .	16
3.3.1. Exported Authenticator Request Characteristics . . .	17
3.4. The CERTIFICATE Frame . . . . .	17
3.4.1. Exported Authenticator Characteristics . . . . .	19
4. Indicating Failures During HTTP-Layer Certificate Authentication . . . . .	19
4.1. Misbehavior . . . . .	20
4.2. Invalid Certificates . . . . .	20
5. Required Domain Certificate Extension . . . . .	20
6. Security Considerations . . . . .	21
6.1. Impersonation . . . . .	21
6.2. Fingerprinting . . . . .	22
6.3. Denial of Service . . . . .	22

6.4.	Persistence of Service . . . . .	22
6.5.	Confusion About State . . . . .	23
7.	IANA Considerations . . . . .	23
7.1.	HTTP/2 Settings . . . . .	23
7.2.	New HTTP/2 Frames . . . . .	24
7.3.	New HTTP/2 Error Codes . . . . .	24
8.	References . . . . .	24
8.1.	Normative References . . . . .	24
8.2.	Informative References . . . . .	25
8.3.	URIs . . . . .	26
Appendix A.	Change Log . . . . .	26
A.1.	Since draft-ietf-httpbis-http2-secondary-certs-04: . . .	26
A.2.	Since draft-ietf-httpbis-http2-secondary-certs-03: . . .	26
A.3.	Since draft-ietf-httpbis-http2-secondary-certs-02: . . .	26
A.4.	Since draft-ietf-httpbis-http2-secondary-certs-01: . . .	26
A.5.	Since draft-ietf-httpbis-http2-secondary-certs-00: . . .	27
A.6.	Since draft-bishop-httpbis-http2-additional-certs-05: . .	27
	Acknowledgements . . . . .	27
	Authors' Addresses . . . . .	27

## 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 used to correlate an HTTP request with a request for a certificate. Since streams are used for individual requests, correlation with a stream is sufficient.

[RFC7540] prohibits renegotiation after any application data has been sent. This completely blocks reactive certificate authentication in HTTP/2 using TLS 1.2. If this restriction were relaxed by an extension or update to HTTP/2, such an identifier could be added to TLS 1.2 by means of an extension to TLS. Unfortunately, many TLS 1.2 implementations do not permit application data to continue during a renegotiation. This is problematic for a multiplexed protocol like HTTP/2.

### 1.3. HTTP-Layer Certificate Authentication

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

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

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

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

Successfully verified Authenticators result in certificate chains, with verified possession of the corresponding private key, which can be supplied into a collection of available certificates. Likewise, descriptions of desired certificates can be supplied into these collections.

Section 2 describes how the feature is employed, defining means to detect support in peers (Section 2.1), make certificates and requests available (Section 2.2), and indicate when streams are blocked waiting on an appropriate certificate (Section 2.3). Section 3 defines the required frame types, which parallel the TLS 1.3 message exchange. Finally, Section 4 defines new error types which can be used to notify peers when the exchange has not been successful.

#### 1.4. Terminology

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

#### 2. Discovering Additional Certificates at the HTTP/2 Layer

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

Otherwise, the server uses a "CERTIFICATE\_REQUEST" frame to describe a class of certificates on stream zero, then uses "CERTIFICATE\_NEEDED" frames to associate these with individual requests. The client responds with a "USE\_CERTIFICATE" frame indicating the certificate which should be used to satisfy the request.

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

## 2.1. Indicating Support for HTTP-Layer Certificate Authentication

Clients and servers that will accept requests for HTTP-layer certificate authentication indicate this using the HTTP/2 "SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH" (0xSETTING-TBD1) and "SETTINGS\_HTTP\_SERVER\_CERT\_AUTH" (0xSETTING-TBD2) settings.

The initial value for both settings is 0, indicating that the peer does not support HTTP-layer certificate authentication. If a peer does support HTTP-layer certificate authentication, one or both of the values is non-zero. "SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH" indicates that servers can use certificates for client authentication, while "SETTINGS\_HTTP\_SERVER\_CERT\_AUTH" indicates that servers are able to offer additional certificates to demonstrate control over other origin hostnames.

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 these settings. The setting values are 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:

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

The value of the exporter is split into two four-byte values. The first four bytes are used for the "SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH" value, while the following four bytes are used for the "SETTINGS\_HTTP\_SERVER\_CERT\_AUTH" value.

Each is converted to a setting value as:

Exporter fragment | 0x80000000

That is, the most significant bit will always be set, regardless of the value of the exporter. Each endpoint will compute the expected

values 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 in the indicated direction.

## 2.2. Making Certificates or Requests Available

When both peers have advertised support for HTTP-layer certificates in a given direction as in Section 2.1, the indicated endpoint can supply additional certificates into the connection at any time. That is, if both endpoints have sent "SETTINGS\_HTTP\_SERVER\_CERT\_AUTH" and validated the value received from the peer, the server may send certificates. If both endpoints have sent "SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH" and validated the value received from the peer, the client may send certificates.

Implementations 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.

Certificates supplied by clients MUST NOT be considered by servers when processing a request unless the client explicitly authorizes their use. Clients MAY send "USE\_CERTIFICATE" frame with the "UNSOLICITED" flag set to indicate that an available certificate should be considered on a new request.

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, a 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 server 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.

If a server has not advertised support for HTTP-layer certificates, fails to provide a requested certificate, or provides a certificate which is unacceptable to the client, the client MUST NOT send any requests for resources in that origin on the current connection. The

client MAY open a new connection in an effort to reach an authoritative server.

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 stream error on the push stream; see section 8.2 of [RFC7540]. To avoid this, servers MUST supply the associated certificates before pushing resources from a different origin.

### 2.3.2. Requiring Additional Client Certificates

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

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

Figure 6: Reactive certificate authentication

If the client does not have the desired certificate, it instead sends an Empty Authenticator, as described in Section 5 of [I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE\_CERTIFICATE" frame which references the Empty Authenticator.

If the client has not advertised support for HTTP-layer certificates, fails to provide a requested certificate, or provides a certificate

the server is unable to verify, 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.

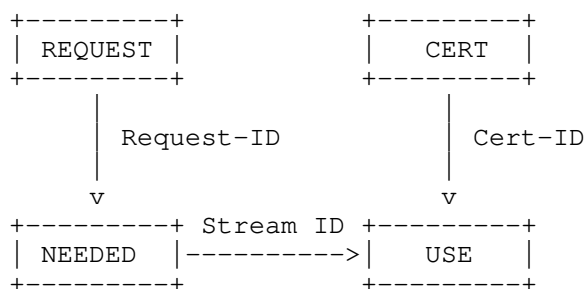


Figure 7: Frame correlation

"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.

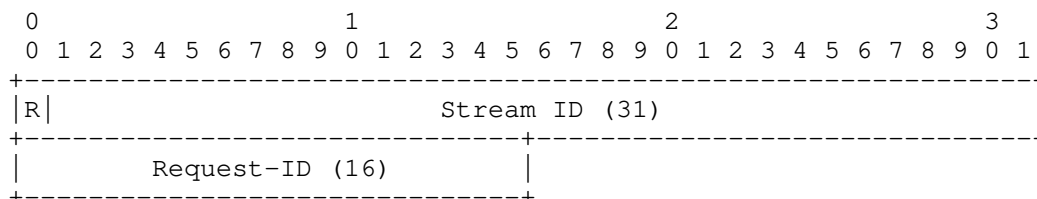


Figure 8: CERTIFICATE\_NEEDED frame payload

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 client which has not advertised the "SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH", or to a server which has not advertised the "SETTINGS\_HTTP\_SERVER\_CERT\_AUTH" setting. An endpoint which receives a "CERTIFICATE\_NEEDED" frame but did not advertise support MAY treat this as a connection error of type "CERTIFICATE\_WITHOUT\_CONSENT".

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:

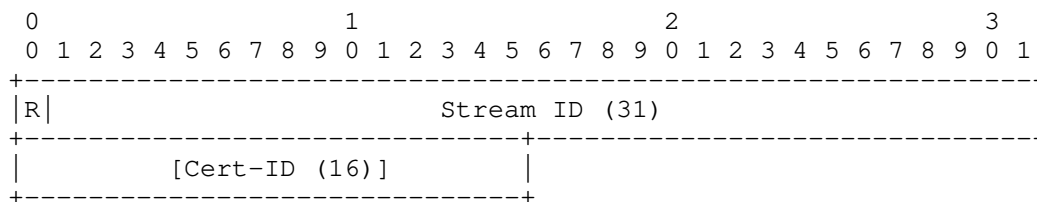


Figure 9: USE\_CERTIFICATE frame payload

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" frame or an unsolicited "USE\_CERTIFICATE" frame which is not the first in

reference to a given stream MUST be treated as a stream error of type "CERTIFICATE\_OVERUSED".

Each "USE\_CERTIFICATE" frame which is not marked as unsolicited is considered to respond in order to the "CERTIFICATE\_NEEDED" frames for the same stream. If a "USE\_CERTIFICATE" frame is received for which a "CERTIFICATE\_NEEDED" frame has not been sent, this MUST be treated as a stream error of type "CERTIFICATE\_OVERUSED".

Receipt of a "USE\_CERTIFICATE" frame with an unknown "Cert-ID" MUST result in a stream error of type "PROTOCOL\_ERROR".

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

### 3.3. The CERTIFICATE\_REQUEST Frame

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

The "CERTIFICATE\_REQUEST" frame SHOULD NOT be sent to a client which has not advertised the "SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH", or to a server which has not advertised the "SETTINGS\_HTTP\_SERVER\_CERT\_AUTH" setting.

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".

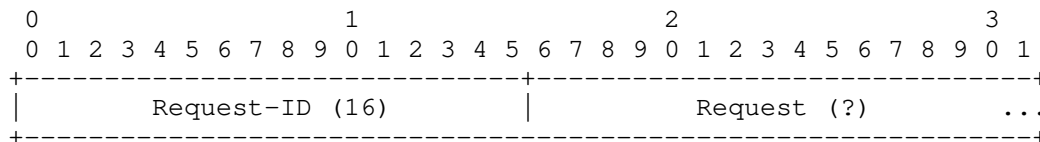


Figure 10: CERTIFICATE\_REQUEST frame payload

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.

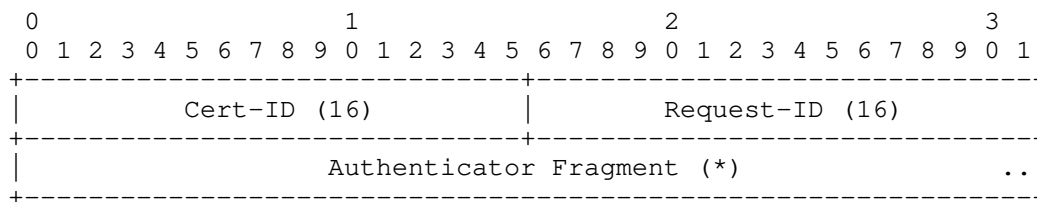


Figure 11: CERTIFICATE frame payload

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:

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

If the authenticator cannot be validated, this SHOULD be treated as a connection error of type "CERTIFICATE\_UNREADABLE".

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.

There are two classes of errors which might be encountered, and they are handled differently.

#### 4.1. Misbehavior

This category of errors could indicate a peer failing to follow restrictions in this document, or might indicate that the connection is not fully secure. These errors are fatal to stream or connection, as appropriate.

`CERTIFICATE_OVERUSED` (0xERROR-TBD1): More certificates were used on a request than were requested

`CERTIFICATE_WITHOUT_CONSENT` (0xERROR-TBD2): A `CERTIFICATE_NEEDED` frame was received by a peer which did not indicate support for this extension.

`CERTIFICATE_UNREADABLE` (0xERROR-TBD3): An exported authenticator could not be validated.

#### 4.2. Invalid Certificates

Unacceptable certificates (expired, revoked, or insufficient to satisfy the request) are not treated as stream or connection errors. This is typically not an indication of a protocol failure. Servers SHOULD process requests with the indicated certificate, likely resulting in a "4XX"-series status code in the response. Clients SHOULD establish a new connection in an attempt to reach an authoritative server.

#### 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. Clients SHOULD consult DNS for hostnames presented in secondary certificates if they would have done so for the same hostname if it were present in the primary 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.

Servers can also learn information about clients using this mechanism. The hostnames a user agent finds interesting and retrieves certificates for might indicate origins the user has previously accessed.

## 6.3. Denial of Service

Failure to provide a certificate for 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.

### 7. IANA Considerations

This draft adds entries in three registries.

The feature negotiation settings 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

The SETTINGS\_HTTP\_CLIENT\_CERT\_AUTH and SETTINGS\_HTTP\_SERVER\_CERT\_AUTH settings are registered in the "HTTP/2 Settings" registry established in [RFC7540].

Name	Code	Initial Value	Specification
HTTP_CLIENT_CERT_AUTH	0xSETTING-TBD1	0	Section 2.1
HTTP_SERVER_CERT_AUTH	0xSETTING-TBD2	0	Section 2.1

## 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.

Frame Type	Code	Specification
CERTIFICATE_NEEDED	0xFRAME-TBD1	Section 3.1
CERTIFICATE_REQUEST	0xFRAME-TBD2	Section 3.3
CERTIFICATE	0xFRAME-TBD3	Section 3.4
USE_CERTIFICATE	0xFRAME-TBD4	Section 3.2

## 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.

Name	Code	Specification
CERTIFICATE_OVERUSED	0xERROR-TBD1	Section 4
CERTIFICATE_WITHOUT_CONSENT	0xERROR-TBD2	Section 4
CERTIFICATE_UNREADABLE	0xERROR-TBD3	Section 4

## 8. References

### 8.1. Normative References

- [I-D.ietf-tls-exported-authenticator]  
Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-11 (work in progress), December 2019.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6844] Hallam-Baker, P. and R. Stradling, "DNS Certification Authority Authorization (CAA) Resource Record", RFC 6844, DOI 10.17487/RFC6844, January 2013, <<https://www.rfc-editor.org/info/rfc6844>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7540] Belshé, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

## 8.2. Informative References

- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, DOI 10.17487/RFC5705, March 2010, <<https://www.rfc-editor.org/info/rfc5705>>.
- [RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.

[RFC8336] Nottingham, M. and E. Nygren, "The ORIGIN HTTP/2 Frame", RFC 8336, DOI 10.17487/RFC8336, March 2018, <<https://www.rfc-editor.org/info/rfc8336>>.

### 8.3. URIs

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

[2] <http://httpwg.github.io/>

[3] <https://github.com/httpwg/http-extensions/labels/secondary-certs>

## 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-04:

Editorial updates only.

### A.2. Since draft-ietf-httpbis-http2-secondary-certs-03:

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

### A.3. Since draft-ietf-httpbis-http2-secondary-certs-02:

Editorial updates only.

### A.4. Since draft-ietf-httpbis-http2-secondary-certs-01:

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

A.5. Since draft-ietf-httpbis-http2-secondary-certs-00:

- o All frames sent on stream zero; replaced "AUTOMATIC\_USE" on "CERTIFICATE" with "UNSOLICITED" on "USE\_CERTIFICATE". (#482, #566)
- o Use Exported Requests from the TLS Exported Authenticators draft; eliminate facilities for expressing certificate requirements in "CERTIFICATE\_REQUEST" frame. (#481)

A.6. Since draft-bishop-httpbis-http2-additional-certs-05:

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

#### Acknowledgements

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

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

#### Authors' Addresses

Mike Bishop  
Akamai

Email: mbishop@evequefou.be

Nick Sullivan  
Cloudflare

Email: nick@cloudflare.com

Martin Thomson  
Mozilla

Email: martin.thomson@gmail.com

HTTP  
Internet-Draft  
Updates: 7234 (if approved)  
Intended status: Standards Track  
Expires: May 5, 2020

M. Nottingham  
Fastly  
November 2, 2019

HTTP Representation Variants  
draft-ietf-httpbis-variants-06

Abstract

This specification introduces an alternative way to select a HTTP response from a cache based upon its request headers, 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](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> [1].

Working Group information can be found at <https://httpwg.github.io/> [2]; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/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 May 5, 2020.

## 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 . . . . .	5
2. The "Variants" HTTP Header Field . . . . .	5
2.1. Relationship to Vary . . . . .	7
3. The "Variant-Key" HTTP Header Field . . . . .	7
4. Cache Behaviour . . . . .	9
4.1. Compute Possible Keys . . . . .	10
4.2. Check Vary . . . . .	11
4.3. Example of Cache Behaviour . . . . .	11
4.3.1. A Variant Missing From the Cache . . . . .	12
4.3.2. Variants That Don't Overlap the Client's Request . .	13
5. Origin Server Behaviour . . . . .	13
5.1. Examples . . . . .	14
5.1.1. Single Variant . . . . .	14
5.1.2. Multiple Variants . . . . .	15
5.1.3. Partial Coverage . . . . .	15
6. Defining Content Negotiation Using Variants . . . . .	16
7. IANA Considerations . . . . .	16
8. Security Considerations . . . . .	17
9. References . . . . .	17
9.1. Normative References . . . . .	17
9.2. Informative References . . . . .	18
9.3. URIs . . . . .	18
Appendix A. Variants for Existing Content Negotiation Mechanisms	19
A.1. Accept . . . . .	19
A.2. Accept-Encoding . . . . .	20
A.3. Accept-Language . . . . .	20
A.4. Cookie . . . . .	21
Acknowledgements . . . . .	22
Author's Address . . . . .	23



## 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. Their values collectively form the secondary cache key for a given response.

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 infrequent implementation in cache, and divergence of behaviours when it is.

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. Effectively, it allows the specification of a request header field to define how it affects the secondary cache key.

When this specification is in use, the example above might become:

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

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

[English content]

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

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

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

### 1.1. Notational Conventions

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

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

Additionally, it uses the "field-name" rule from [RFC7230], "type", "subtype", "content-coding" and "language-range" from [RFC7231], and "cookie-name" from [RFC6265].

### 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 a representation of the values that are available for each.

Variants is a Structured Header Dictionary (Section 3.2 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

```
Variants      = sh-dict
```

Each member-name represents the field-name of a request header that is part of the secondary cache key; each member-value is an inner-list of strings or tokens that convey representations of potential values for that header field, hereafter referred to as "available-values".

If Structured Header parsing fails or a member's value does have the structure outlined above, the client MUST treat the representation as having no Variants header field.

Note that an available-value that is a token is interpreted as a string containing the same characters, and vice versa.

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 dictionaries, 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 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 one or more sets of available-values that identify the secondary cache key(s) that the response it occurs within are associated with.

Variant-Key is a Structured Header List (Section 3.1 of [I-D.ietf-httpbis-header-structure]) whose members are inner-lists of strings or tokens. Its ABNF is:

```
Variant-Key      = sh-list
```

Each member MUST be an inner-list, and MUST itself have the same number of members as there are members of the representation's Variants header field. If not, the client MUST treat the representation as having no Variant-Key header field.

Each member identifies a list of available-values corresponding to the header field-names in the Variants header field, thereby identifying a secondary cache key that can be used with this response. These available-values do not need to explicitly appear in the Variants header field; they can be interpreted by the algorithm specific to processing that field. For example, Accept-Encoding defines an implicit "identity" available-value (Appendix A.2).

Each inner-list member is treated as identifying an available-value for the corresponding variant-axis' field-name. Any list-member that is a token is interpreted as a string containing the same characters.

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. 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 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 field-values, after being combined as allowed by Section 3.2.2 of [RFC7230]), 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, 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 incoming-request (a mapping of field-names to field-values, after being combined as allowed by Section 3.2.2 of [RFC7230]), and 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:

```
[
  ["fr", "en"]           // prefers French, will accept English
  ["gzip", "identity"] // prefers gzip encoding, will accept identity
]
```

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:

- o MUST define a request header field that advertises the clients preferences or capabilities, whose field-name SHOULD begin with "Accept-".
- o MUST define the syntax of an available-value that will occur in Variants and Variant-Key.
- o 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]:

- o Header field name: Variants
- o Applicable protocol: http
- o Status: standard

- o Author/Change Controller: IETF
- o Specification document(s): [this document]
- o Related information:

This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

- o Header field name: Variant-Key
- o Applicable protocol: http
- o Status: standard
- o Author/Change Controller: IETF
- o Specification document(s): [this document]
- o Related information:

## 8. Security Considerations

If the number or advertised characteristics of the representations available for a resource are considered sensitive, the Variants header by its nature will leak them.

Note that the Variants header is not a commitment to make representations of a certain nature available; the runtime behaviour of the server always overrides hints like Variants.

## 9. References

### 9.1. Normative References

- [I-D.ietf-httpbis-header-structure]  
Nottingham, M. and P. Kamp, "Structured Headers for HTTP", draft-ietf-httpbis-header-structure-13 (work in progress), August 2019.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4647] Phillips, A. and M. Davis, "Matching of Language Tags", BCP 47, RFC 4647, DOI 10.17487/RFC4647, September 2006, <<https://www.rfc-editor.org/info/rfc4647>>.

- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <<https://www.rfc-editor.org/info/rfc7234>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

## 9.2. Informative References

- [I-D.ietf-httpbis-client-hints] Grigorik, I., "HTTP Client Hints", draft-ietf-httpbis-client-hints-07 (work in progress), March 2019.
- [RFC2295] Holtman, K. and A. Mutz, "Transparent Content Negotiation in HTTP", RFC 2295, DOI 10.17487/RFC2295, March 1998, <<https://www.rfc-editor.org/info/rfc2295>>.
- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <<https://www.rfc-editor.org/info/rfc3864>>.

## 9.3. URIs

- [1] <https://lists.w3.org/Archives/Public/ietf-http-wg/>
- [2] <https://httpwg.github.io/>



[3] <https://github.com/httpwg/http-extensions/labels/variants>

[4] <https://github.com/mnot/variants-toy>

## 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.

#### A.4. Cookie

This section defines variant handling for the Cookie request header ([RFC6265]).

This syntax of an available-value for Cookie is:

cookie-available-value = cookie-name

To perform content negotiation for Cookie given a request-value and available-values:

1. Let cookies-available be an empty list.
2. For each available-value of available-values:
  1. Parse request-value as a Cookie header field [RFC6265] and let request-cookie-value be the cookie-value corresponding to a cookie with a cookie-name that matches available-value. If no match is found, continue to the next available-value.
  2. append request-cookie-value to cookies-available.
3. Return cookies-available.

A simple example is allowing a page designed for users that aren't logged in (denoted by the "logged\_in" cookie-name) to be cached:

```
Variants: Cookie=(logged_in)
Variant-Key: (0)
Vary: Cookie
```

Here, a cache that implements Variants will only use this response to satisfy requests with "Cookie: logged\_in=0". Caches that don't implement Variants will vary the response on all Cookie headers.

Or, consider this example:

```
Variants: Cookie=(user_priority)
Variant-Key: (silver), ("bronze")
Vary: Cookie
```

Here, the "user\_priority" cookie-name allows requests from "gold" users to be separated from "silver" and "bronze" ones; this response is only served to the latter two.

It is possible to target a response to a single user; for example:

```
Variants: Cookie=(user_id)
Variant-Key: (some_person)
Vary: Cookie
```

Here, only the "some\_person" "user\_id" will have this response served to them again.

Note that if more than one cookie-name serves as a cache key, they'll need to be listed in separate Variants members, like this:

```
Variants: Cookie=(user_priority), Cookie=(user_region)
Variant-Key: (gold europe)
Vary: Cookie
```

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

QUIC  
Internet-Draft  
Intended status: Standards Track  
Expires: 6 August 2021

M. Bishop, Ed.  
Akamai  
2 February 2021

Hypertext Transfer Protocol Version 3 (HTTP/3)  
draft-ietf-quic-http-34

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to HTTP/3.

DO NOT DEPLOY THIS VERSION OF HTTP

DO NOT DEPLOY THIS VERSION OF HTTP/3 UNTIL IT IS IN AN RFC. This version is still a work in progress. For trial deployments, please use earlier versions.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list ([quic@ietf.org](mailto:quic@ietf.org)), which is archived at [https://mailarchive.ietf.org/arch/search/?email\\_list=quic](https://mailarchive.ietf.org/arch/search/?email_list=quic).

Working Group information can be found at <https://github.com/quicwg>; source code and issues list for this draft can be found at <https://github.com/quicwg/base-drafts/labels/-http>.

Status of This Memo

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

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

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

This Internet-Draft will expire on 6 August 2021.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

## Table of Contents

1. Introduction . . . . .	4
1.1. Prior versions of HTTP . . . . .	5
1.2. Delegation to QUIC . . . . .	5
2. HTTP/3 Protocol Overview . . . . .	5
2.1. Document Organization . . . . .	6
2.2. Conventions and Terminology . . . . .	7
3. Connection Setup and Management . . . . .	8
3.1. Discovering an HTTP/3 Endpoint . . . . .	8
3.1.1. HTTP Alternative Services . . . . .	9
3.1.2. Other Schemes . . . . .	10
3.2. Connection Establishment . . . . .	10
3.3. Connection Reuse . . . . .	11
4. HTTP Request Lifecycle . . . . .	12
4.1. HTTP Message Exchanges . . . . .	12
4.1.1. Field Formatting and Compression . . . . .	14
4.1.2. Request Cancellation and Rejection . . . . .	17
4.1.3. Malformed Requests and Responses . . . . .	18
4.2. The CONNECT Method . . . . .	19
4.3. HTTP Upgrade . . . . .	21
4.4. Server Push . . . . .	21
5. Connection Closure . . . . .	23
5.1. Idle Connections . . . . .	23
5.2. Connection Shutdown . . . . .	24
5.3. Immediate Application Closure . . . . .	26
5.4. Transport Closure . . . . .	26
6. Stream Mapping and Usage . . . . .	27
6.1. Bidirectional Streams . . . . .	27
6.2. Unidirectional Streams . . . . .	28
6.2.1. Control Streams . . . . .	29
6.2.2. Push Streams . . . . .	30

6.2.3. Reserved Stream Types . . . . .	30
7. HTTP Framing Layer . . . . .	31
7.1. Frame Layout . . . . .	32
7.2. Frame Definitions . . . . .	32
7.2.1. DATA . . . . .	32
7.2.2. HEADERS . . . . .	33
7.2.3. CANCEL_PUSH . . . . .	33
7.2.4. SETTINGS . . . . .	35
7.2.5. PUSH_PROMISE . . . . .	38
7.2.6. GOAWAY . . . . .	39
7.2.7. MAX_PUSH_ID . . . . .	40
7.2.8. Reserved Frame Types . . . . .	41
8. Error Handling . . . . .	41
8.1. HTTP/3 Error Codes . . . . .	42
9. Extensions to HTTP/3 . . . . .	43
10. Security Considerations . . . . .	44
10.1. Server Authority . . . . .	44
10.2. Cross-Protocol Attacks . . . . .	44
10.3. Intermediary Encapsulation Attacks . . . . .	45
10.4. Cacheability of Pushed Responses . . . . .	45
10.5. Denial-of-Service Considerations . . . . .	45
10.5.1. Limits on Field Section Size . . . . .	46
10.5.2. CONNECT Issues . . . . .	47
10.6. Use of Compression . . . . .	47
10.7. Padding and Traffic Analysis . . . . .	48
10.8. Frame Parsing . . . . .	48
10.9. Early Data . . . . .	49
10.10. Migration . . . . .	49
10.11. Privacy Considerations . . . . .	49
11. IANA Considerations . . . . .	49
11.1. Registration of HTTP/3 Identification String . . . . .	50
11.2. New Registries . . . . .	50
11.2.1. Frame Types . . . . .	50
11.2.2. Settings Parameters . . . . .	52
11.2.3. Error Codes . . . . .	53
11.2.4. Stream Types . . . . .	55
12. References . . . . .	56
12.1. Normative References . . . . .	56
12.2. Informative References . . . . .	57
Appendix A. Considerations for Transitioning from HTTP/2 . . . . .	58
A.1. Streams . . . . .	59
A.2. HTTP Frame Types . . . . .	60
A.2.1. Prioritization Differences . . . . .	60
A.2.2. Field Compression Differences . . . . .	60
A.2.3. Flow Control Differences . . . . .	61
A.2.4. Guidance for New Frame Type Definitions . . . . .	61
A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types . . . . .	61
A.3. HTTP/2 SETTINGS Parameters . . . . .	62



A.4. HTTP/2 Error Codes . . . . .	64
A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors . . . . .	65
Appendix B. Change Log . . . . .	65
B.1. Since draft-ietf-quic-http-32 . . . . .	65
B.2. Since draft-ietf-quic-http-31 . . . . .	66
B.3. Since draft-ietf-quic-http-30 . . . . .	66
B.4. Since draft-ietf-quic-http-29 . . . . .	66
B.5. Since draft-ietf-quic-http-28 . . . . .	66
B.6. Since draft-ietf-quic-http-27 . . . . .	66
B.7. Since draft-ietf-quic-http-26 . . . . .	66
B.8. Since draft-ietf-quic-http-25 . . . . .	66
B.9. Since draft-ietf-quic-http-24 . . . . .	67
B.10. Since draft-ietf-quic-http-23 . . . . .	67
B.11. Since draft-ietf-quic-http-22 . . . . .	67
B.12. Since draft-ietf-quic-http-21 . . . . .	68
B.13. Since draft-ietf-quic-http-20 . . . . .	68
B.14. Since draft-ietf-quic-http-19 . . . . .	69
B.15. Since draft-ietf-quic-http-18 . . . . .	69
B.16. Since draft-ietf-quic-http-17 . . . . .	69
B.17. Since draft-ietf-quic-http-16 . . . . .	70
B.18. Since draft-ietf-quic-http-15 . . . . .	70
B.19. Since draft-ietf-quic-http-14 . . . . .	70
B.20. Since draft-ietf-quic-http-13 . . . . .	70
B.21. Since draft-ietf-quic-http-12 . . . . .	71
B.22. Since draft-ietf-quic-http-11 . . . . .	71
B.23. Since draft-ietf-quic-http-10 . . . . .	71
B.24. Since draft-ietf-quic-http-09 . . . . .	71
B.25. Since draft-ietf-quic-http-08 . . . . .	72
B.26. Since draft-ietf-quic-http-07 . . . . .	72
B.27. Since draft-ietf-quic-http-06 . . . . .	72
B.28. Since draft-ietf-quic-http-05 . . . . .	72
B.29. Since draft-ietf-quic-http-04 . . . . .	72
B.30. Since draft-ietf-quic-http-03 . . . . .	72
B.31. Since draft-ietf-quic-http-02 . . . . .	73
B.32. Since draft-ietf-quic-http-01 . . . . .	73
B.33. Since draft-ietf-quic-http-00 . . . . .	73
B.34. Since draft-shade-quic-http2-mapping-00 . . . . .	74
Acknowledgments . . . . .	74
Author's Address . . . . .	75

## 1. Introduction

HTTP semantics ([SEMANTICS]) are used for a broad range of services on the Internet. These semantics have most commonly been used with HTTP/1.1 and HTTP/2. HTTP/1.1 has been used over a variety of transport and session layers, while HTTP/2 has been used primarily with TLS over TCP. HTTP/3 supports the same semantics over a new transport protocol, QUIC.

### 1.1. Prior versions of HTTP

HTTP/1.1 ([HTTP11]) uses whitespace-delimited text fields to convey HTTP messages. While these exchanges are human-readable, using whitespace for message formatting leads to parsing complexity and excessive tolerance of variant behavior.

Because HTTP/1.1 does not include a multiplexing layer, multiple TCP connections are often used to service requests in parallel. However, that has a negative impact on congestion control and network efficiency, since TCP does not share congestion control across multiple connections.

HTTP/2 ([HTTP2]) introduced a binary framing and multiplexing layer to improve latency without modifying the transport layer. However, because the parallel nature of HTTP/2's multiplexing is not visible to TCP's loss recovery mechanisms, a lost or reordered packet causes all active transactions to experience a stall regardless of whether that transaction was directly impacted by the lost packet.

### 1.2. Delegation to QUIC

The QUIC transport protocol incorporates stream multiplexing and per-stream flow control, similar to that provided by the HTTP/2 framing layer. By providing reliability at the stream level and congestion control across the entire connection, QUIC has the capability to improve the performance of HTTP compared to a TCP mapping. QUIC also incorporates TLS 1.3 ([TLS13]) at the transport layer, offering comparable confidentiality and integrity to running TLS over TCP, with the improved connection setup latency of TCP Fast Open ([TFO]).

This document defines HTTP/3, a mapping of HTTP semantics over the QUIC transport protocol, drawing heavily on the design of HTTP/2. HTTP/3 relies on QUIC to provide confidentiality and integrity protection of data; peer authentication; and reliable, in-order, per-stream delivery. While delegating stream lifetime and flow control issues to QUIC, a binary framing similar to the HTTP/2 framing is used on each stream. Some HTTP/2 features are subsumed by QUIC, while other features are implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [HTTP2].

## 2. HTTP/3 Protocol Overview

HTTP/3 provides a transport for HTTP semantics using the QUIC transport protocol and an internal framing layer similar to HTTP/2.

Once a client knows that an HTTP/3 server exists at a certain endpoint, it opens a QUIC connection. QUIC provides protocol negotiation, stream-based multiplexing, and flow control. Discovery of an HTTP/3 endpoint is described in Section 3.1.

Within each stream, the basic unit of HTTP/3 communication is a frame (Section 7.2). Each frame type serves a different purpose. For example, HEADERS and DATA frames form the basis of HTTP requests and responses (Section 4.1). Frames that apply to the entire connection are conveyed on a dedicated control stream.

Multiplexing of requests is performed using the QUIC stream abstraction, described in Section 2 of [QUIC-TRANSPORT]. Each request-response pair consumes a single QUIC stream. Streams are independent of each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

Server push is an interaction mode introduced in HTTP/2 ([HTTP2]) that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. Several HTTP/3 frames are used to manage server push, such as PUSH\_PROMISE, MAX\_PUSH\_ID, and CANCEL\_PUSH.

As in HTTP/2, request and response fields are compressed for transmission. Because HPACK ([HPACK]) relies on in-order transmission of compressed field sections (a guarantee not provided by QUIC), HTTP/3 replaces HPACK with QPACK ([QPACK]). QPACK uses separate unidirectional streams to modify and track field table state, while encoded field sections refer to the state of the table without modifying it.

## 2.1. Document Organization

The following sections provide a detailed overview of the lifecycle of an HTTP/3 connection:

- \* Connection Setup and Management (Section 3) covers how an HTTP/3 endpoint is discovered and an HTTP/3 connection is established.
- \* HTTP Request Lifecycle (Section 4) describes how HTTP semantics are expressed using frames.
- \* Connection Closure (Section 5) describes how HTTP/3 connections are terminated, either gracefully or abruptly.

The details of the wire protocol and interactions with the transport are described in subsequent sections:

- \* Stream Mapping and Usage (Section 6) describes the way QUIC streams are used.
- \* HTTP Framing Layer (Section 7) describes the frames used on most streams.
- \* Error Handling (Section 8) describes how error conditions are handled and expressed, either on a particular stream or for the connection as a whole.

Additional resources are provided in the final sections:

- \* Extensions to HTTP/3 (Section 9) describes how new capabilities can be added in future documents.
- \* A more detailed comparison between HTTP/2 and HTTP/3 can be found in Appendix A.

## 2.2. Conventions and Terminology

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

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].

The following terms are used:

**abort:** An abrupt termination of a connection or stream, possibly due to an error condition.

**client:** The endpoint that initiates an HTTP/3 connection. Clients send HTTP requests and receive HTTP responses.

**connection:** A transport-layer connection between two endpoints, using QUIC as the transport protocol.

**connection error:** An error that affects the entire HTTP/3 connection.

**endpoint:** Either the client or server of the connection.

**frame:** The smallest unit of communication on a stream in HTTP/3, consisting of a header and a variable-length sequence of bytes structured according to the frame type.

Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC CONNECTION\_CLOSE frames." References without this preface refer to frames defined in Section 7.2.

**HTTP/3 connection:** A QUIC connection where the negotiated application protocol is HTTP/3.

**peer:** An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

**receiver:** An endpoint that is receiving frames.

**sender:** An endpoint that is transmitting frames.

**server:** The endpoint that accepts an HTTP/3 connection. Servers receive HTTP requests and send HTTP responses.

**stream:** A bidirectional or unidirectional bytestream provided by the QUIC transport. All streams within an HTTP/3 connection can be considered "HTTP/3 streams," but multiple stream types are defined within HTTP/3.

**stream error:** An application-level error on the individual stream.

The term "content" is defined in Section 6.4 of [SEMANTICS].

Finally, the terms "resource", "message", "user agent", "origin server", "gateway", "intermediary", "proxy", and "tunnel" are defined in Section 3 of [SEMANTICS].

Packet diagrams in this document use the format defined in Section 1.3 of [QUIC-TRANSPORT] to illustrate the order and size of fields.

### 3. Connection Setup and Management

#### 3.1. Discovering an HTTP/3 Endpoint

HTTP relies on the notion of an authoritative response: a response that has been determined to be the most appropriate response for that request given the state of the target resource at the time of response message origination by (or at the direction of) the origin server identified within the target URI. Locating an authoritative server for an HTTP URI is discussed in Section 4.3 of [SEMANTICS].

The "https" scheme associates authority with possession of a certificate that the client considers to be trustworthy for the host identified by the authority component of the URI. Upon receiving a server certificate in the TLS handshake, the client **MUST** verify that the certificate is an acceptable match for the URI's origin server using the process described in Section 4.3.4 of [SEMANTICS]. If the certificate cannot be verified with respect to the URI's origin server, the client **MUST NOT** consider the server authoritative for that origin.

A client **MAY** attempt access to a resource with an "https" URI by resolving the host identifier to an IP address, establishing a QUIC connection to that address on the indicated port (including validation of the server certificate as described above), and sending an HTTP/3 request message targeting the URI to the server over that secured connection. Unless some other mechanism is used to select HTTP/3, the token "h3" is used in the Application Layer Protocol Negotiation (ALPN; see [RFC7301]) extension during the TLS handshake.

Connectivity problems (e.g., blocking UDP) can result in QUIC connection establishment failure; clients **SHOULD** attempt to use TCP-based versions of HTTP in this case.

Servers **MAY** serve HTTP/3 on any UDP port; an alternative service advertisement always includes an explicit port, and URIs contain either an explicit port or a default port associated with the scheme.

#### 3.1.1. HTTP Alternative Services

An HTTP origin can advertise the availability of an equivalent HTTP/3 endpoint via the Alt-Svc HTTP response header field or the HTTP/2 ALTSVC frame ([ALTSVC]), using the "h3" ALPN token.

For example, an origin could indicate in an HTTP response that HTTP/3 was available on UDP port 50781 at the same hostname by including the following header field:

```
Alt-Svc: h3=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/3 support, a client **MAY** attempt to establish a QUIC connection to the indicated host and port; if this connection is successful, the client can send HTTP requests using the mapping described in this document.

### 3.1.2. Other Schemes

Although HTTP is independent of the transport protocol, the "http" scheme associates authority with the ability to receive TCP connections on the indicated port of whatever host is identified within the authority component. Because HTTP/3 does not use TCP, HTTP/3 cannot be used for direct access to the authoritative server for a resource identified by an "http" URI. However, protocol extensions such as [ALTSVC] permit the authoritative server to identify other services that are also authoritative and that might be reachable over HTTP/3.

Prior to making requests for an origin whose scheme is not "https", the client **MUST** ensure the server is willing to serve that scheme. For origins whose scheme is "http", an experimental method to accomplish this is described in [RFC8164]. Other mechanisms might be defined for various schemes in the future.

### 3.2. Connection Establishment

HTTP/3 relies on QUIC version 1 as the underlying transport. The use of other QUIC transport versions with HTTP/3 **MAY** be defined by future specifications.

QUIC version 1 uses TLS version 1.3 or greater as its handshake protocol. HTTP/3 clients **MUST** support a mechanism to indicate the target host to the server during the TLS handshake. If the server is identified by a domain name ([DNS-TERMS]), clients **MUST** send the Server Name Indication (SNI; [RFC6066]) TLS extension unless an alternative mechanism to indicate the target host is used.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/3 support is indicated by selecting the ALPN token "h3" in the TLS handshake. Support for other application-layer protocols **MAY** be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP/3-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 7.2.4) **MUST** be sent by each endpoint as the initial frame of their respective HTTP control stream; see Section 6.2.1.

### 3.3. Connection Reuse

HTTP/3 connections are persistent across multiple requests. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. To use an existing connection for a new origin, clients MUST validate the certificate presented by the server for the new origin server using the process described in Section 4.3.4 of [SEMANTICS]. This implies that clients will need to retain the server certificate and any additional information needed to verify that certificate; clients which do not do so will be unable to reuse the connection for additional origins.

If the certificate is not acceptable with regard to the new origin for any reason, the connection MUST NOT be reused and a new connection SHOULD be established for the new origin. If the reason the certificate cannot be verified might apply to other origins already associated with the connection, the client SHOULD re-validate the server certificate for those origins. For instance, if validation of a certificate fails because the certificate has expired or been revoked, this might be used to invalidate all other origins for which that certificate was used to establish authority.

Clients SHOULD NOT open more than one HTTP/3 connection to a given IP address and UDP port, where the IP address and port might be derived from a URI, a selected alternative service ([ALTSVC]), a configured proxy, or name resolution of any of these. A client MAY open multiple HTTP/3 connections to the same IP address and UDP port using different transport or TLS configurations but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open HTTP/3 connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the HTTP/3 connection, the terminating endpoint SHOULD first send a GOAWAY frame (Section 5.2) so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

A server that does not wish clients to reuse HTTP/3 connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request; see Section 7.4 of [SEMANTICS].



## 4. HTTP Request Lifecycle

### 4.1. HTTP Message Exchanges

A client sends an HTTP request on a request stream, which is a client-initiated bidirectional QUIC stream; see Section 6.1. A client **MUST** send only a single request on a given stream. A server sends zero or more interim HTTP responses on the same stream as the request, followed by a single final HTTP response, as detailed below. See Section 15 of [SEMANTICS] for a description of interim and final HTTP responses.

Pushed responses are sent on a server-initiated unidirectional QUIC stream; see Section 6.2.2. A server sends zero or more interim HTTP responses, followed by a single final HTTP response, in the same manner as a standard response. Push is described in more detail in Section 4.4.

On a given stream, receipt of multiple requests or receipt of an additional HTTP response following a final HTTP response **MUST** be treated as malformed (Section 4.1.3).

An HTTP message (request or response) consists of:

1. the header section, sent as a single HEADERS frame (see Section 7.2.2),
2. optionally, the content, if present, sent as a series of DATA frames (see Section 7.2.1), and
3. optionally, the trailer section, if present, sent as a single HEADERS frame.

Header and trailer sections are described in Sections 6.3 and 6.5 of [SEMANTICS]; the content is described in Section 6.4 of [SEMANTICS].

Receipt of an invalid sequence of frames **MUST** be treated as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8. In particular, a DATA frame before any HEADERS frame, or a HEADERS or DATA frame after the trailing HEADERS frame, is considered invalid. Other frame types, especially unknown frame types, might be permitted subject to their own rules; see Section 9.

A server MAY send one or more PUSH\_PROMISE frames (Section 7.2.5) before, after, or interleaved with the frames of a response message. These PUSH\_PROMISE frames are not part of the response; see Section 4.4 for more details. PUSH\_PROMISE frames are not permitted on push streams; a pushed response that includes PUSH\_PROMISE frames MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

Frames of unknown types (Section 9), including reserved frames (Section 7.2.8) MAY be sent on a request or push stream before, after, or interleaved with other frames described in this section.

The HEADERS and PUSH\_PROMISE frames might reference updates to the QPACK dynamic table. While these updates are not directly part of the message exchange, they must be received and processed before the message can be consumed. See Section 4.1.1 for more details.

Transfer codings (see Section 6.1 of [HTTP11]) are not defined for HTTP/3; the Transfer-Encoding header field MUST NOT be used.

A response MAY consist of multiple messages when and only when one or more interim responses (1xx; see Section 15.2 of [SEMANTICS]) precede a final response to the same request. Interim responses do not contain content or trailer sections.

An HTTP request/response exchange fully consumes a client-initiated bidirectional QUIC stream. After sending a request, a client MUST close the stream for sending. Unless using the CONNECT method (see Section 4.2), clients MUST NOT make stream closure dependent on receiving a response to their request. After sending a final response, the server MUST close the stream for sending. At this point, the QUIC stream is fully closed.

When a stream is closed, this indicates the end of the final HTTP message. Because some messages are large or unbounded, endpoints SHOULD begin processing partial HTTP messages once enough of the message has been received to make progress. If a client-initiated stream terminates without enough of the HTTP message to provide a complete response, the server SHOULD abort its response stream with the error code H3\_REQUEST\_INCOMPLETE; see Section 8.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When the server does not need to receive the remainder of the request, it MAY abort reading the request stream, send a complete response, and cleanly close the sending part of the stream. The error code H3\_NO\_ERROR SHOULD be used when requesting that the client stop sending on the

request stream. Clients **MUST NOT** discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. If the server sends a partial or complete response but does not abort reading the request, clients **SHOULD** continue sending the body of the request and close the stream normally.

#### 4.1.1. Field Formatting and Compression

HTTP messages carry metadata as a series of key-value pairs called HTTP fields; see Sections 6.3 and 6.5 of [SEMANTICS]. For a listing of registered HTTP fields, see the "Hypertext Transfer Protocol (HTTP) Field Name Registry" maintained at <https://www.iana.org/assignments/http-fields/>.

*\*Note:* This registry will not exist until [SEMANTICS] is approved. *\*RFC Editor\**, please remove this note prior to publication.

Field names are strings containing a subset of ASCII characters. Properties of HTTP field names and values are discussed in more detail in Section 5.1 of [SEMANTICS]. As in HTTP/2, characters in field names **MUST** be converted to lowercase prior to their encoding. A request or response containing uppercase characters in field names **MUST** be treated as malformed (Section 4.1.3).

Like HTTP/2, HTTP/3 does not use the Connection header field to indicate connection-specific fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint **MUST NOT** generate an HTTP/3 field section containing connection-specific fields; any message containing connection-specific fields **MUST** be treated as malformed (Section 4.1.3).

The only exception to this is the TE header field, which **MAY** be present in an HTTP/3 request header; when it is, it **MUST NOT** contain any value other than "trailers".

An intermediary transforming an HTTP/1.x message to HTTP/3 **MUST** remove connection-specific header fields as discussed in Section 7.6.1 of [SEMANTICS], or their messages will be treated by other HTTP/3 endpoints as malformed (Section 4.1.3).

##### 4.1.1.1. Pseudo-Header Fields

Like HTTP/2, HTTP/3 employs a series of pseudo-header fields where the field name begins with the ':' character (ASCII 0x3a). These pseudo-header fields convey the target URI, the method of the request, and the status code for the response.

Pseudo-header fields are not HTTP fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document; however, an extension could negotiate a modification of this restriction; see Section 9.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailer sections. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 4.1.3).

All pseudo-header fields MUST appear in the header section before regular header fields. Any request or response that contains a pseudo-header field that appears in a header section after a regular header field MUST be treated as malformed (Section 4.1.3).

The following pseudo-header fields are defined for requests:

**":method":** Contains the HTTP method (Section 9 of [SEMANTICS])

**":scheme":** Contains the scheme portion of the target URI (Section 3.1 of [URI])

**":scheme"** is not restricted to URIs with scheme "http" and "https". A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

See Section 3.1.2 for guidance on using a scheme other than "https".

**":authority":** Contains the authority portion of the target URI (Section 3.2 of [URI]). The authority MUST NOT include the deprecated "userinfo" subcomponent for URIs of scheme "http" or "https".

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field MUST be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form; see Section 7.1 of [SEMANTICS]. Clients that generate HTTP/3 requests directly SHOULD use the **":authority"** pseudo-header field instead of the Host field. An intermediary that converts an HTTP/3 request to HTTP/1.1 MUST create a Host field if one is not present in a request by copying the value of the **":authority"** pseudo-header field.

`:path`: Contains the path and query parts of the target URI (the "path-absolute" production and optionally a '?' character followed by the "query" production; see Sections 3.3 and 3.4 of [URI]). A request in asterisk form includes the value '\*' for the `:path` pseudo-header field.

This pseudo-header field MUST NOT be empty for "http" or "https" URIs; "http" or "https" URIs that do not contain a path component MUST include a value of '/'. The exception to this rule is an OPTIONS request for an "http" or "https" URI that does not include a path component; these MUST include a `:path` pseudo-header field with a value of '\*'; see Section 7.1 of [SEMANTICS].

All HTTP/3 requests MUST include exactly one value for the `:method`, `:scheme`, and `:path` pseudo-header fields, unless it is a CONNECT request; see Section 4.2.

If the `:scheme` pseudo-header field identifies a scheme that has a mandatory authority component (including "http" and "https"), the request MUST contain either an `:authority` pseudo-header field or a "Host" header field. If these fields are present, they MUST NOT be empty. If both fields are present, they MUST contain the same value. If the scheme does not have a mandatory authority component and none is provided in the request target, the request MUST NOT contain the `:authority` pseudo-header or "Host" header fields.

An HTTP request that omits mandatory pseudo-header fields or contains invalid values for those pseudo-header fields is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

For responses, a single `:status` pseudo-header field is defined that carries the HTTP status code; see Section 15 of [SEMANTICS]. This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 4.1.3).

HTTP/3 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

#### 4.1.1.2. Field Compression

[QPACK] describes a variation of HPACK that gives an encoder some control over how much head-of-line blocking can be caused by compression. This allows an encoder to balance compression efficiency with latency. HTTP/3 uses QPACK to compress header and trailer sections, including the pseudo-header fields present in the header section.

To allow for better compression efficiency, the "Cookie" field ([RFC6265]) MAY be split into separate field lines, each with one or more cookie-pairs, before compression. If a decompressed field section contains multiple cookie field lines, these MUST be concatenated into a single byte string using the two-byte delimiter of 0x3b, 0x20 (the ASCII string "; ") before being passed into a context other than HTTP/2 or HTTP/3, such as an HTTP/1.1 connection, or a generic HTTP server application.

#### 4.1.1.3. Header Size Constraints

An HTTP/3 implementation MAY impose a limit on the maximum size of the message header it will accept on an individual HTTP message. A server that receives a larger header section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process. The size of a field list is calculated based on the uncompressed size of fields, including the length of the name and value in bytes plus an overhead of 32 bytes for each field.

If an implementation wishes to advise its peer of this limit, it can be conveyed as a number of bytes in the SETTINGS\_MAX\_FIELD\_SECTION\_SIZE parameter. An implementation that has received this parameter SHOULD NOT send an HTTP message header that exceeds the indicated size, as the peer will likely refuse to process it. However, an HTTP message can traverse one or more intermediaries before reaching the origin server; see Section 3.7 of [SEMANTICS]. Because this limit is applied separately by each implementation which processes the message, messages below this limit are not guaranteed to be accepted.

#### 4.1.2. Request Cancellation and Rejection

Once a request stream has been opened, the request MAY be cancelled by either endpoint. Clients cancel requests if the response is no longer of interest; servers cancel requests if they are unable to or choose not to respond. When possible, it is RECOMMENDED that servers send an HTTP response with an appropriate status code rather than canceling a request it has already begun processing.

Implementations SHOULD cancel requests by abruptly terminating any directions of a stream that are still open. This means resetting the sending parts of streams and aborting reading on receiving parts of streams; see Section 2.4 of [QUIC-TRANSPORT].

When the server cancels a request without performing any application processing, the request is considered "rejected." The server SHOULD abort its response stream with the error code H3\_REQUEST\_REJECTED. In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result. The client can treat requests rejected by the server as though they had never been sent at all, thereby allowing them to be retried later.

Servers MUST NOT use the H3\_REQUEST\_REJECTED error code for requests that were partially or fully processed. When a server abandons a response after partial processing, it SHOULD abort its response stream with the error code H3\_REQUEST\_CANCELLED.

Client SHOULD use the error code H3\_REQUEST\_CANCELLED to cancel requests. Upon receipt of this error code, a server MAY abruptly terminate the response using the error code H3\_REQUEST\_REJECTED if no processing was performed. Clients MUST NOT use the H3\_REQUEST\_REJECTED error code, except when a server has requested closure of the request stream with this error code.

If a stream is canceled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Only idempotent actions such as GET, PUT, or DELETE can be safely retried; a client SHOULD NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are idempotent independent of the method or some means to detect that the original request was never applied. See Section 9.2.2 of [SEMANTICS] for more details.

#### 4.1.3. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of frames but is invalid due to:

- \* the presence of prohibited fields or pseudo-header fields,
- \* the absence of mandatory pseudo-header fields,
- \* invalid values for pseudo-header fields,

- \* pseudo-header fields after fields,
- \* an invalid sequence of HTTP messages,
- \* the inclusion of uppercase field names, or
- \* the inclusion of invalid characters in field names or values.

A request or response that is defined as having content when it contains a Content-Length header field (Section 6.4.1 of [SEMANTICS]), is malformed if the value of a Content-Length header field does not equal the sum of the DATA frame lengths received. A response that is defined as never having content, even when a Content-Length is present, can have a non-zero Content-Length field even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 8) of type H3\_MESSAGE\_ERROR.

For malformed requests, a server MAY send an HTTP response indicating the error prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

#### 4.2. The CONNECT Method

The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target; see Section 9.3.6 of [SEMANTICS]. It is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2 and HTTP/3, the CONNECT method is used to establish a tunnel over a single stream.

A CONNECT request MUST be constructed as follows:

- \* The ":method" pseudo-header field is set to "CONNECT"
- \* The ":scheme" and ":path" pseudo-header fields are omitted



- \* The ":authority" pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests; see Section 7.1 of [SEMANTICS])

The request stream remains open at the end of the request to carry the data to be transferred. A CONNECT request that does not conform to these restrictions is malformed; see Section 4.1.3.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in Section 15.3 of [SEMANTICS].

All DATA frames on the stream correspond to data sent or received on the TCP connection. The payload of any DATA frame sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

Once the CONNECT method has completed, only DATA frames are permitted to be sent on the stream. Extension frames MAY be used if specifically permitted by the definition of the extension. Receipt of any other known frame type MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will close the send stream that it sends to the client. TCP connections that remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT close a stream for sending while they still expect to receive data from the target of the CONNECT.

A TCP connection error is signaled by abruptly terminating the stream. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type H3\_CONNECT\_ERROR; see Section 8. Correspondingly, if a proxy detects an error with the stream or the QUIC connection, it MUST close the TCP connection. If the underlying TCP implementation permits it, the proxy SHOULD send a TCP segment with the RST bit set.

Since CONNECT creates a tunnel to an arbitrary server, proxies that support CONNECT SHOULD restrict its use to a set of known ports or a list of safe request targets; see Section 9.3.6 of [SEMANTICS] for more detail.

#### 4.3. HTTP Upgrade

HTTP/3 does not support the HTTP Upgrade mechanism (Section 7.8 of [SEMANTICS]) or 101 (Switching Protocols) informational status code (Section 15.2.2 of [SEMANTICS]).

#### 4.4. Server Push

Server push is an interaction mode that permits a server to push a request-response exchange to a client in anticipation of the client making the indicated request. This trades off network usage against a potential latency gain. HTTP/3 server push is similar to what is described in Section 8.2 of [HTTP2], but uses different mechanisms.

Each server push is assigned a unique Push ID by the server. The Push ID is used to refer to the push in various contexts throughout the lifetime of the HTTP/3 connection.

The Push ID space begins at zero, and ends at a maximum value set by the MAX\_PUSH\_ID frame; see Section 7.2.7. In particular, a server is not able to push until after the client sends a MAX\_PUSH\_ID frame. A client sends MAX\_PUSH\_ID frames to control the number of pushes that a server can promise. A server SHOULD use Push IDs sequentially, beginning from zero. A client MUST treat receipt of a push stream as a connection error of type H3\_ID\_ERROR (Section 8) when no MAX\_PUSH\_ID frame has been sent or when the stream references a Push ID that is greater than the maximum Push ID.

The Push ID is used in one or more PUSH\_PROMISE frames (Section 7.2.5) that carry the header section of the request message. These frames are sent on the request stream that generated the push. This allows the server push to be associated with a client request. When the same Push ID is promised on multiple request streams, the decompressed request field sections MUST contain the same fields in the same order, and both the name and the value in each field MUST be identical.

The Push ID is then included with the push stream that ultimately fulfills those promises; see Section 6.2.2. The push stream identifies the Push ID of the promise that it fulfills, then contains a response to the promised request as described in Section 4.1.

Finally, the Push ID can be used in CANCEL\_PUSH frames; see Section 7.2.3. Clients use this frame to indicate they do not wish to receive a promised resource. Servers use this frame to indicate they will not be fulfilling a previous promise.

Not all requests can be pushed. A server MAY push requests that have the following properties:

- \* cacheable; see Section 9.2.3 of [SEMANTICS]
- \* safe; see Section 9.2.1 of [SEMANTICS]
- \* does not include a request body or trailer section

The server MUST include a value in the ":authority" pseudo-header field for which the server is authoritative. If the client has not yet validated the connection for the origin indicated by the pushed request, it MUST perform the same verification process it would do before sending a request for that origin on the connection; see Section 3.3. If this verification fails, the client MUST NOT consider the server authoritative for that origin.

Clients SHOULD send a CANCEL\_PUSH frame upon receipt of a PUSH\_PROMISE frame carrying a request that is not cacheable, is not known to be safe, that indicates the presence of a request body, or for which it does not consider the server authoritative. Any corresponding responses MUST NOT be used or cached.

Each pushed response is associated with one or more client requests. The push is associated with the request stream on which the PUSH\_PROMISE frame was received. The same server push can be associated with additional client requests using a PUSH\_PROMISE frame with the same Push ID on multiple request streams. These associations do not affect the operation of the protocol, but MAY be considered by user agents when deciding how to use pushed resources.

Ordering of a PUSH\_PROMISE frame in relation to certain parts of the response is important. The server SHOULD send PUSH\_PROMISE frames prior to sending HEADERS or DATA frames that reference the promised responses. This reduces the chance that a client requests a resource that will be pushed by the server.

Due to reordering, push stream data can arrive before the corresponding PUSH\_PROMISE frame. When a client receives a new push stream with an as-yet-unknown Push ID, both the associated client request and the pushed request header fields are unknown. The client can buffer the stream data in expectation of the matching PUSH\_PROMISE. The client can use stream flow control (see Section 4.1 of [QUIC-TRANSPORT]) to limit the amount of data a server may commit to the pushed stream.

Push stream data can also arrive after a client has canceled a push. In this case, the client can abort reading the stream with an error code of H3\_REQUEST\_CANCELLED. This asks the server not to transfer additional data and indicates that it will be discarded upon receipt.

Pushed responses that are cacheable (see Section 3 of [CACHING]) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present; see Section 5.2.2.3 of [CACHING]) at the time the pushed response is received.

Pushed responses that are not cacheable MUST NOT be stored by any HTTP cache. They MAY be made available to the application separately.

## 5. Connection Closure

Once established, an HTTP/3 connection can be used for many requests and responses over time until the connection is closed. Connection closure can happen in any of several different ways.

### 5.1. Idle Connections

Each QUIC endpoint declares an idle timeout during the handshake. If the QUIC connection remains idle (no packets received) for longer than this duration, the peer will assume that the connection has been closed. HTTP/3 implementations will need to open a new HTTP/3 connection for new requests if the existing connection has been idle for longer than the idle timeout negotiated during the QUIC handshake, and SHOULD do so if approaching the idle timeout; see Section 10.1 of [QUIC-TRANSPORT].

HTTP clients are expected to request that the transport keep connections open while there are responses outstanding for requests or server pushes, as described in Section 10.1.2 of [QUIC-TRANSPORT]. If the client is not expecting a response from the server, allowing an idle connection to time out is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY

maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers. Servers SHOULD NOT actively keep connections open.

## 5.2. Connection Shutdown

Even when a connection is not idle, either endpoint can decide to stop using the connection and initiate a graceful connection close. Endpoints initiate the graceful shutdown of an HTTP/3 connection by sending a GOAWAY frame (Section 7.2.6). The GOAWAY frame contains an identifier that indicates to the receiver the range of requests or pushes that were or might be processed in this connection. The server sends a client-initiated bidirectional Stream ID; the client sends a Push ID (Section 4.4). Requests or pushes with the indicated identifier or greater are rejected (Section 4.1.2) by the sender of the GOAWAY. This identifier MAY be zero if no requests or pushes were processed.

The information in the GOAWAY frame enables a client and server to agree on which requests or pushes were accepted prior to the shutdown of the HTTP/3 connection. Upon sending a GOAWAY frame, the endpoint SHOULD explicitly cancel (see Section 4.1.2 and Section 7.2.3) any requests or pushes that have identifiers greater than or equal to that indicated, in order to clean up transport state for the affected streams. The endpoint SHOULD continue to do so as more requests or pushes arrive.

Endpoints MUST NOT initiate new requests or promise new pushes on the connection after receipt of a GOAWAY frame from the peer. Clients MAY establish a new connection to send additional requests.

Some requests or pushes might already be in transit:

- \* Upon receipt of a GOAWAY frame, if the client has already sent requests with a Stream ID greater than or equal to the identifier contained in the GOAWAY frame, those requests will not be processed. Clients can safely retry unprocessed requests on a different HTTP connection. A client that is unable to retry requests loses all requests that are in flight when the server closes the connection.

Requests on Stream IDs less than the Stream ID in a GOAWAY frame from the server might have been processed; their status cannot be known until a response is received, the stream is reset individually, another GOAWAY is received with a lower Stream ID than that of the request in question, or the connection terminates.

Servers MAY reject individual requests on streams below the indicated ID if these requests were not processed.

- \* If a server receives a GOAWAY frame after having promised pushes with a Push ID greater than or equal to the identifier contained in the GOAWAY frame, those pushes will not be accepted.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a request has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

An endpoint MAY send multiple GOAWAY frames indicating different identifiers, but the identifier in each frame MUST NOT be greater than the identifier in any previous frame, since clients might already have retried unprocessed requests on another HTTP connection. Receiving a GOAWAY containing a larger identifier than previously received MUST be treated as a connection error of type H3\_ID\_ERROR; see Section 8.

An endpoint that is attempting to gracefully shut down a connection can send a GOAWAY frame with a value set to the maximum possible value ( $2^{62}-4$  for servers,  $2^{62}-1$  for clients). This ensures that the peer stops creating new requests or pushes. After allowing time for any in-flight requests or pushes to arrive, the endpoint can send another GOAWAY frame indicating which requests or pushes it might accept before the end of the connection. This ensures that a connection can be cleanly shut down without losing requests.

A client has more flexibility in the value it chooses for the Push ID in a GOAWAY that it sends. A value of  $2^{62}-1$  indicates that the server can continue fulfilling pushes that have already been promised. A smaller value indicates the client will reject pushes with Push IDs greater than or equal to this value. Like the server, the client MAY send subsequent GOAWAY frames so long as the specified Push ID is no greater than any previously sent value.

Even when a GOAWAY indicates that a given request or push will not be processed or accepted upon receipt, the underlying transport resources still exist. The endpoint that initiated these requests can cancel them to clean up transport state.

Once all accepted requests and pushes have been processed, the endpoint can permit the connection to become idle, or MAY initiate an immediate closure of the connection. An endpoint that completes a graceful shutdown SHOULD use the H3\_NO\_ERROR error code when closing the connection.

If a client has consumed all available bidirectional stream IDs with requests, the server need not send a GOAWAY frame, since the client is unable to make further requests.

### 5.3. Immediate Application Closure

An HTTP/3 implementation can immediately close the QUIC connection at any time. This results in sending a QUIC CONNECTION\_CLOSE frame to the peer indicating that the application layer has terminated the connection. The application error code in this frame indicates to the peer why the connection is being closed. See Section 8 for error codes that can be used when closing a connection in HTTP/3.

Before closing the connection, a GOAWAY frame MAY be sent to allow the client to retry some requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION\_CLOSE frame improves the chances of the frame being received by clients.

If there are open streams that have not been explicitly closed, they are implicitly closed when the connection is closed; see Section 10.2 of [QUIC-TRANSPORT].

### 5.4. Transport Closure

For various reasons, the QUIC transport could indicate to the application layer that the connection has terminated. This might be due to an explicit closure by the peer, a transport-level error, or a change in network topology that interrupts connectivity.

If a connection terminates without a GOAWAY frame, clients MUST assume that any request that was sent, whether in whole or in part, might have been processed.

## 6. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. In version 1 of QUIC, the stream data containing HTTP frames is carried by QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. The transport layer buffers and orders received stream data, exposing a reliable byte stream to the application. Although QUIC permits out-of-order delivery within a stream, HTTP/3 does not make use of this feature.

QUIC streams can be either unidirectional, carrying data only from initiator to receiver, or bidirectional. Streams can be initiated by either the client or the server. For more detail on QUIC streams, see Section 2 of [QUIC-TRANSPORT].

When HTTP fields and data are sent over QUIC, the QUIC layer handles most of the stream management. HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction or to the entire HTTP/3 connection context.

### 6.1. Bidirectional Streams

All client-initiated bidirectional streams are used for HTTP requests and responses. A bidirectional stream ensures that the response can be readily correlated with the request. These streams are referred to as request streams.

This means that the client's first request occurs on QUIC stream 0, with subsequent requests on stream 4, 8, and so on. In order to permit these streams to open, an HTTP/3 server SHOULD configure non-zero minimum values for the number of permitted streams and the initial stream flow control window. So as to not unnecessarily limit parallelism, at least 100 request streams SHOULD be permitted at a time.

HTTP/3 does not use server-initiated bidirectional streams, though an extension could define a use for these streams. Clients MUST treat receipt of a server-initiated bidirectional stream as a connection error of type H3\_STREAM\_CREATION\_ERROR (Section 8) unless such an extension has been negotiated.



## 6.2. Unidirectional Streams

Unidirectional streams, in either direction, are used for a range of purposes. The purpose is indicated by a stream type, which is sent as a variable-length integer at the start of the stream. The format and structure of data that follows this integer is determined by the stream type.

```
Unidirectional Stream Header {  
    Stream Type (i),  
}
```

Figure 1: Unidirectional Stream Header

Two stream types are defined in this document: control streams (Section 6.2.1) and push streams (Section 6.2.2). [QPACK] defines two additional stream types. Other stream types can be defined by extensions to HTTP/3; see Section 9 for more details. Some stream types are reserved (Section 6.2.3).

The performance of HTTP/3 connections in the early phase of their lifetime is sensitive to the creation and exchange of data on unidirectional streams. Endpoints that excessively restrict the number of streams or the flow control window of these streams will increase the chance that the remote peer reaches the limit early and becomes blocked. In particular, implementations should consider that remote peers may wish to exercise reserved stream behavior (Section 6.2.3) with some of the unidirectional streams they are permitted to use. To avoid blocking, the transport parameters sent by both clients and servers **MUST** allow the peer to create at least one unidirectional stream for the HTTP control stream plus the number of unidirectional streams required by mandatory extensions (three being the minimum number required for the base HTTP/3 protocol and QPACK), and **SHOULD** provide at least 1,024 bytes of flow control credit to each stream.

Note that an endpoint is not required to grant additional credits to create more unidirectional streams if its peer consumes all the initial credits before creating the critical unidirectional streams. Endpoints **SHOULD** create the HTTP control stream as well as the unidirectional streams required by mandatory extensions (such as the QPACK encoder and decoder streams) first, and then create additional streams as allowed by their peer.

If the stream header indicates a stream type that is not supported by the recipient, the remainder of the stream cannot be consumed as the semantics are unknown. Recipients of unknown stream types MAY abort reading of the stream with an error code of `H3_STREAM_CREATION_ERROR` or a reserved error code (Section 8.1), but MUST NOT consider such streams to be a connection error of any kind.

Implementations MAY send stream types before knowing whether the peer supports them. However, stream types that could modify the state or semantics of existing protocol components, including QPACK or other extensions, MUST NOT be sent until the peer is known to support them.

A sender can close or reset a unidirectional stream unless otherwise specified. A receiver MUST tolerate unidirectional streams being closed or reset prior to the reception of the unidirectional stream header.

#### 6.2.1. Control Streams

A control stream is indicated by a stream type of `0x00`. Data on this stream consists of HTTP/3 frames, as defined in Section 7.2.

Each side MUST initiate a single control stream at the beginning of the connection and send its SETTINGS frame as the first frame on this stream. If the first frame of the control stream is any other frame type, this MUST be treated as a connection error of type `H3_MISSING_SETTINGS`. Only one control stream per peer is permitted; receipt of a second stream claiming to be a control stream MUST be treated as a connection error of type `H3_STREAM_CREATION_ERROR`. The sender MUST NOT close the control stream, and the receiver MUST NOT request that the sender close the control stream. If either control stream is closed at any point, this MUST be treated as a connection error of type `H3_CLOSED_CRITICAL_STREAM`. Connection errors are described in Section 8.

Because the contents of the control stream are used to manage the behavior of other streams, endpoints SHOULD provide enough flow control credit to keep the peer's control stream from becoming blocked.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon as it is able. Depending on whether 0-RTT is available on the QUIC connection, either client or server might be able to send stream data first.

### 6.2.2. Push Streams

Server push is an optional feature introduced in HTTP/2 that allows a server to initiate a response before a request has been made. See Section 4.4 for more details.

A push stream is indicated by a stream type of 0x01, followed by the Push ID of the promise that it fulfills, encoded as a variable-length integer. The remaining data on this stream consists of HTTP/3 frames, as defined in Section 7.2, and fulfills a promised server push by zero or more interim HTTP responses followed by a single final HTTP response, as defined in Section 4.1. Server push and Push IDs are described in Section 4.4.

Only servers can push; if a server receives a client-initiated push stream, this MUST be treated as a connection error of type H3\_STREAM\_CREATION\_ERROR; see Section 8.

```
Push Stream Header {  
    Stream Type (i) = 0x01,  
    Push ID (i),  
}
```

Figure 2: Push Stream Header

Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type H3\_ID\_ERROR; see Section 8.

### 6.2.3. Reserved Stream Types

Stream types of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored. These streams have no semantics, and can be sent when application-layer padding is desired. They MAY also be sent on connections where no data is currently being transferred. Endpoints MUST NOT consider these streams to have any meaning upon receipt.

The payload and length of the stream are selected in any manner the sending implementation chooses. When sending a reserved stream type, the implementation MAY either terminate the stream cleanly or reset it. When resetting the stream, either the H3\_NO\_ERROR error code or a reserved error code (Section 8.1) SHOULD be used.

## 7. HTTP Framing Layer

HTTP frames are carried on QUIC streams, as described in Section 6. HTTP/3 defines three stream types: control stream, request stream, and push stream. This section describes HTTP/3 frame formats and their permitted stream types; see Table 1 for an overview. A comparison between HTTP/2 and HTTP/3 frames is provided in Appendix A.2.

Frame	Control Stream	Request Stream	Push Stream	Section
DATA	No	Yes	Yes	Section 7.2.1
HEADERS	No	Yes	Yes	Section 7.2.2
CANCEL_PUSH	Yes	No	No	Section 7.2.3
SETTINGS	Yes (1)	No	No	Section 7.2.4
PUSH_PROMISE	No	Yes	No	Section 7.2.5
GOAWAY	Yes	No	No	Section 7.2.6
MAX_PUSH_ID	Yes	No	No	Section 7.2.7
Reserved	Yes	Yes	Yes	Section 7.2.8

Table 1: HTTP/3 Frames and Stream Type Overview

The SETTINGS frame can only occur as the first frame of a Control stream; this is indicated in Table 1 with a (1). Specific guidance is provided in the relevant section.

Note that, unlike QUIC frames, HTTP/3 frames can span multiple packets.

## 7.1. Frame Layout

All frames have the following format:

```
HTTP/3 Frame Format {  
    Type (i),  
    Length (i),  
    Frame Payload (...),  
}
```

Figure 3: HTTP/3 Frame Format

A frame includes the following fields:

Type: A variable-length integer that identifies the frame type.

Length: A variable-length integer that describes the length in bytes of the Frame Payload.

Frame Payload: A payload, the semantics of which are determined by the Type field.

Each frame's payload MUST contain exactly the fields identified in its description. A frame payload that contains additional bytes after the identified fields or a frame payload that terminates before the end of the identified fields MUST be treated as a connection error of type `H3_FRAME_ERROR`; see Section 8. In particular, redundant length encodings MUST be verified to be self-consistent; see Section 10.8.

When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error of type `H3_FRAME_ERROR`; see Section 8. Streams that terminate abruptly may be reset at any point in a frame.

## 7.2. Frame Definitions

### 7.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of bytes associated with HTTP request or response content.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on a control stream, the recipient MUST respond with a connection error of type `H3_FRAME_UNEXPECTED`; see Section 8.

```
DATA Frame {  
    Type (i) = 0x0,  
    Length (i),  
    Data (...),  
}
```

Figure 4: DATA Frame

#### 7.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry an HTTP field section, encoded using QPACK. See [QPACK] for more details.

```
HEADERS Frame {  
    Type (i) = 0x1,  
    Length (i),  
    Encoded Field Section (...),  
}
```

Figure 5: HEADERS Frame

HEADERS frames can only be sent on request or push streams. If a HEADERS frame is received on a control stream, the recipient MUST respond with a connection error (Section 8) of type H3\_FRAME\_UNEXPECTED.

#### 7.2.3. CANCEL\_PUSH

The CANCEL\_PUSH frame (type=0x3) is used to request cancellation of a server push prior to the push stream being received. The CANCEL\_PUSH frame identifies a server push by Push ID (see Section 4.4), encoded as a variable-length integer.

When a client sends CANCEL\_PUSH, it is indicating that it does not wish to receive the promised resource. The server SHOULD abort sending the resource, but the mechanism to do so depends on the state of the corresponding push stream. If the server has not yet created a push stream, it does not create one. If the push stream is open, the server SHOULD abruptly terminate that stream. If the push stream has already ended, the server MAY still abruptly terminate the stream or MAY take no action.

A server sends CANCEL\_PUSH to indicate that it will not be fulfilling a promise which was previously sent. The client cannot expect the corresponding promise to be fulfilled, unless it has already received and processed the promised response. Regardless of whether a push stream has been opened, a server SHOULD send a CANCEL\_PUSH frame when it determines that promise will not be fulfilled. If a stream has already been opened, the server can abort sending on the stream with an error code of H3\_REQUEST\_CANCELLED.

Sending a CANCEL\_PUSH frame has no direct effect on the state of existing push streams. A client SHOULD NOT send a CANCEL\_PUSH frame when it has already received a corresponding push stream. A push stream could arrive after a client has sent a CANCEL\_PUSH frame, because a server might not have processed the CANCEL\_PUSH. The client SHOULD abort reading the stream with an error code of H3\_REQUEST\_CANCELLED.

A CANCEL\_PUSH frame is sent on the control stream. Receiving a CANCEL\_PUSH frame on a stream other than the control stream MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED.

```
CANCEL_PUSH Frame {  
  Type (i) = 0x3,  
  Length (i),  
  Push ID (i),  
}
```

Figure 6: CANCEL\_PUSH Frame

The CANCEL\_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled; see Section 4.4. If a CANCEL\_PUSH frame is received that references a Push ID greater than currently allowed on the connection, this MUST be treated as a connection error of type H3\_ID\_ERROR.

If the client receives a CANCEL\_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH\_PROMISE frame due to reordering. If a server receives a CANCEL\_PUSH frame for a Push ID that has not yet been mentioned by a PUSH\_PROMISE frame, this MUST be treated as a connection error of type H3\_ID\_ERROR.

#### 7.2.4. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior. Individually, a SETTINGS parameter can also be referred to as a "setting"; the identifier and value of each setting parameter can be referred to as a "setting identifier" and a "setting value".

SETTINGS frames always apply to an entire HTTP/3 connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of each control stream (see Section 6.2.1) by each peer, and MUST NOT be sent subsequently. If an endpoint receives a second SETTINGS frame on the control stream, the endpoint MUST respond with a connection error of type H3\_FRAME\_UNEXPECTED.

SETTINGS frames MUST NOT be sent on any stream other than the control stream. If an endpoint receives a SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type H3\_FRAME\_UNEXPECTED.

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer that can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - each peer uses SETTINGS to advertise a set of supported values. The definition of the setting would describe how each peer combines the two sets to conclude which choice will be used. SETTINGS does not provide a mechanism to identify when the choice takes effect.

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume a very large response field section, while servers are more cautious about request size.

The same setting identifier MUST NOT occur more than once in the SETTINGS frame. A receiver MAY treat the presence of duplicate setting identifiers as a connection error of type H3\_SETTINGS\_ERROR.

The payload of a SETTINGS frame consists of zero or more parameters. Each parameter consists of a setting identifier and a value, both encoded as QUIC variable-length integers.



```
Setting {  
    Identifier (i),  
    Value (i),  
}  
  
SETTINGS Frame {  
    Type (i) = 0x4,  
    Length (i),  
    Setting (...) ...,  
}
```

Figure 7: SETTINGS Frame

An implementation MUST ignore any parameter with an identifier it does not understand.

#### 7.2.4.1. Defined SETTINGS Parameters

The following settings are defined in HTTP/3:

**SETTINGS\_MAX\_FIELD\_SECTION\_SIZE (0x6):** The default value is unlimited. See Section 4.1.1.3 for usage.

Setting identifiers of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown identifiers be ignored. Such settings have no defined meaning. Endpoints SHOULD include at least one such setting in their SETTINGS frame. Endpoints MUST NOT consider such settings to have any meaning upon receipt.

Because the setting has no defined meaning, the value of the setting can be any value the implementation selects.

Setting identifiers which were defined in [HTTP2] where there is no corresponding HTTP/3 setting have also been reserved (Section 11.2.2). These reserved settings MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3\_SETTINGS\_ERROR.

Additional settings can be defined by extensions to HTTP/3; see Section 9 for more details.

#### 7.2.4.2. Initialization

An HTTP implementation MUST NOT send frames or requests that would be invalid based on its current understanding of the peer's settings.

All settings begin at an initial value. Each endpoint SHOULD use these initial values to send messages before the peer's SETTINGS frame has arrived, as packets carrying the settings can be lost or delayed. When the SETTINGS frame arrives, any settings are changed to their new values.

This removes the need to wait for the SETTINGS frame before sending messages. Endpoints MUST NOT require any data to be received from the peer prior to sending the SETTINGS frame; settings MUST be sent as soon as the transport is ready to send data.

For servers, the initial value of each client setting is the default value.

For clients using a 1-RTT QUIC connection, the initial value of each server setting is the default value. 1-RTT keys will always become available prior to the packet containing SETTINGS being processed by QUIC, even if the server sends SETTINGS immediately. Clients SHOULD NOT wait indefinitely for SETTINGS to arrive before sending requests, but SHOULD process received datagrams in order to increase the likelihood of processing SETTINGS before sending the first request.

When a 0-RTT QUIC connection is being used, the initial value of each server setting is the value used in the previous session. Clients SHOULD store the settings the server provided in the HTTP/3 connection where resumption information was provided, but MAY opt not to store settings in certain cases (e.g., if the session ticket is received before the SETTINGS frame). A client MUST comply with stored settings -- or default values, if no values are stored -- when attempting 0-RTT. Once a server has provided new settings, clients MUST comply with those values.

A server can remember the settings that it advertised, or store an integrity-protected copy of the values in the ticket and recover the information when accepting 0-RTT data. A server uses the HTTP/3 settings values in determining whether to accept 0-RTT data. If the server cannot determine that the settings remembered by a client are compatible with its current settings, it MUST NOT accept 0-RTT data. Remembered settings are compatible if a client complying with those settings would not violate the server's current settings.

A server MAY accept 0-RTT and subsequently provide different settings in its SETTINGS frame. If 0-RTT data is accepted by the server, its SETTINGS frame MUST NOT reduce any limits or alter any values that might be violated by the client with its 0-RTT data. The server MUST include all settings that differ from their default values. If a server accepts 0-RTT but then sends settings that are not compatible with the previously specified settings, this MUST be treated as a

connection error of type `H3_SETTINGS_ERROR`. If a server accepts 0-RTT but then sends a `SETTINGS` frame that omits a setting value that the client understands (apart from reserved setting identifiers) that was previously specified to have a non-default value, this **MUST** be treated as a connection error of type `H3_SETTINGS_ERROR`.

#### 7.2.5. `PUSH_PROMISE`

The `PUSH_PROMISE` frame (type=0x5) is used to carry a promised request header section from server to client on a request stream, as in HTTP/2.

```
PUSH_PROMISE Frame {  
    Type (i) = 0x5,  
    Length (i),  
    Push ID (i),  
    Encoded Field Section (..),  
}
```

Figure 8: `PUSH_PROMISE` Frame

The payload consists of:

**Push ID:** A variable-length integer that identifies the server push operation. A Push ID is used in push stream headers (Section 4.4) and `CANCEL_PUSH` frames (Section 7.2.3).

**Encoded Field Section:** QPACK-encoded request header fields for the promised response. See [QPACK] for more details.

A server **MUST NOT** use a Push ID that is larger than the client has provided in a `MAX_PUSH_ID` frame (Section 7.2.7). A client **MUST** treat receipt of a `PUSH_PROMISE` frame that contains a larger Push ID than the client has advertised as a connection error of `H3_ID_ERROR`.

A server **MAY** use the same Push ID in multiple `PUSH_PROMISE` frames. If so, the decompressed request header sets **MUST** contain the same fields in the same order, and both the name and the value in each field **MUST** be exact matches. Clients **SHOULD** compare the request header sections for resources promised multiple times. If a client receives a Push ID that has already been promised and detects a mismatch, it **MUST** respond with a connection error of type `H3_GENERAL_PROTOCOL_ERROR`. If the decompressed field sections match exactly, the client **SHOULD** associate the pushed content with each stream on which a `PUSH_PROMISE` frame was received.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH\_PROMISE frame that uses a Push ID that they have already consumed and discarded are forced to ignore the promise.

If a PUSH\_PROMISE frame is received on the control stream, the client MUST respond with a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

A client MUST NOT send a PUSH\_PROMISE frame. A server MUST treat the receipt of a PUSH\_PROMISE frame as a connection error of type H3\_FRAME\_UNEXPECTED; see Section 8.

See Section 4.4 for a description of the overall server push mechanism.

#### 7.2.6. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of an HTTP/3 connection by either endpoint. GOAWAY allows an endpoint to stop accepting new requests or pushes while still finishing processing of previously received requests and pushes. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

```
GOAWAY Frame {  
    Type (i) = 0x7,  
    Length (i),  
    Stream ID/Push ID (...),  
}
```

Figure 9: GOAWAY Frame

The GOAWAY frame is always sent on the control stream. In the server to client direction, it carries a QUIC Stream ID for a client-initiated bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type H3\_ID\_ERROR.

In the client to server direction, the GOAWAY frame carries a Push ID encoded as a variable-length integer.

The GOAWAY frame applies to the entire connection, not a specific stream. A client **MUST** treat a GOAWAY frame on a stream other than the control stream as a connection error of type `H3_FRAME_UNEXPECTED`; see Section 8.

See Section 5.2 for more information on the use of the GOAWAY frame.

#### 7.2.7. MAX\_PUSH\_ID

The `MAX_PUSH_ID` frame (type=0xd) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in `PUSH_PROMISE` and `CANCEL_PUSH` frames. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit maintained by the QUIC transport.

The `MAX_PUSH_ID` frame is always sent on the control stream. Receipt of a `MAX_PUSH_ID` frame on any other stream **MUST** be treated as a connection error of type `H3_FRAME_UNEXPECTED`.

A server **MUST NOT** send a `MAX_PUSH_ID` frame. A client **MUST** treat the receipt of a `MAX_PUSH_ID` frame as a connection error of type `H3_FRAME_UNEXPECTED`.

The maximum Push ID is unset when an HTTP/3 connection is created, meaning that a server cannot push until it receives a `MAX_PUSH_ID` frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending `MAX_PUSH_ID` frames as the server fulfills or cancels server pushes.

```
MAX_PUSH_ID Frame {  
    Type (i) = 0xd,  
    Length (i),  
    Push ID (i),  
}
```

Figure 10: `MAX_PUSH_ID` Frame

The `MAX_PUSH_ID` frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use; see Section 4.4. A `MAX_PUSH_ID` frame cannot reduce the maximum Push ID; receipt of a `MAX_PUSH_ID` frame that contains a smaller value than previously received **MUST** be treated as a connection error of type `H3_ID_ERROR`.

### 7.2.8. Reserved Frame Types

Frame types of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown types be ignored (Section 9). These frames have no semantics, and MAY be sent on any stream where frames are allowed to be sent. This enables their use for application-layer padding. Endpoints MUST NOT consider these frames to have any meaning upon receipt.

The payload and length of the frames are selected in any manner the implementation chooses.

Frame types that were used in HTTP/2 where there is no corresponding HTTP/3 frame have also been reserved (Section 11.2.1). These frame types MUST NOT be sent, and their receipt MUST be treated as a connection error of type H3\_FRAME\_UNEXPECTED.

## 8. Error Handling

When a stream cannot be completed successfully, QUIC allows the application to abruptly terminate (reset) that stream and communicate a reason; see Section 2.4 of [QUIC-TRANSPORT]. This is referred to as a "stream error." An HTTP/3 implementation can decide to close a QUIC stream and communicate the type of error. Wire encodings of error codes are defined in Section 8.1. Stream errors are distinct from HTTP status codes which indicate error conditions. Stream errors indicate that the sender did not transfer or consume the full request or response, while HTTP status codes indicate the result of a request that was successfully received.

If an entire connection needs to be terminated, QUIC similarly provides mechanisms to communicate a reason; see Section 5.3 of [QUIC-TRANSPORT]. This is referred to as a "connection error." Similar to stream errors, an HTTP/3 implementation can terminate a QUIC connection and communicate the reason using an error code from Section 8.1.

Although the reasons for closing streams and connections are called "errors," these actions do not necessarily indicate a problem with the connection or either implementation. For example, a stream can be reset if the requested resource is no longer needed.

An endpoint MAY choose to treat a stream error as a connection error under certain circumstances, closing the entire connection in response to a condition on a single stream. Implementations need to consider the impact on outstanding requests before making this choice.

Because new error codes can be defined without negotiation (see Section 9), use of an error code in an unexpected context or receipt of an unknown error code MUST be treated as equivalent to H3\_NO\_ERROR. However, closing a stream can have other effects regardless of the error code; for example, see Section 4.1.

### 8.1. HTTP/3 Error Codes

The following error codes are defined for use when abruptly terminating streams, aborting reading of streams, or immediately closing HTTP/3 connections.

H3\_NO\_ERROR (0x100): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

H3\_GENERAL\_PROTOCOL\_ERROR (0x101): Peer violated protocol requirements in a way that does not match a more specific error code, or endpoint declines to use the more specific error code.

H3\_INTERNAL\_ERROR (0x102): An internal error has occurred in the HTTP stack.

H3\_STREAM\_CREATION\_ERROR (0x103): The endpoint detected that its peer created a stream that it will not accept.

H3\_CLOSED\_CRITICAL\_STREAM (0x104): A stream required by the HTTP/3 connection was closed or reset.

H3\_FRAME\_UNEXPECTED (0x105): A frame was received that was not permitted in the current state or on the current stream.

H3\_FRAME\_ERROR (0x106): A frame that fails to satisfy layout requirements or with an invalid size was received.

H3\_EXCESSIVE\_LOAD (0x107): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

H3\_ID\_ERROR (0x108): A Stream ID or Push ID was used incorrectly, such as exceeding a limit, reducing a limit, or being reused.

H3\_SETTINGS\_ERROR (0x109): An endpoint detected an error in the payload of a SETTINGS frame.

H3\_MISSING\_SETTINGS (0x10a): No SETTINGS frame was received at the beginning of the control stream.

H3\_REQUEST\_REJECTED (0x10b): A server rejected a request without performing any application processing.

H3\_REQUEST\_CANCELLED (0x10c): The request or its response (including pushed response) is cancelled.

H3\_REQUEST\_INCOMPLETE (0x10d): The client's stream terminated without containing a fully-formed request.

H3\_MESSAGE\_ERROR (0x10e): An HTTP message was malformed and cannot be processed.

H3\_CONNECT\_ERROR (0x10f): The TCP connection established in response to a CONNECT request was reset or abnormally closed.

H3\_VERSION\_FALLBACK (0x110): The requested operation cannot be served over HTTP/3. The peer should retry over HTTP/1.1.

Error codes of the format "0x1f \* N + 0x21" for non-negative integer values of N are reserved to exercise the requirement that unknown error codes be treated as equivalent to H3\_NO\_ERROR (Section 9). Implementations SHOULD select an error code from this space with some probability when they would have sent H3\_NO\_ERROR.

## 9. Extensions to HTTP/3

HTTP/3 permits extension of the protocol. Within the limitations described in this section, protocol extensions can be used to provide additional services or alter any aspect of the protocol. Extensions are effective only within the scope of a single HTTP/3 connection.

This applies to the protocol elements defined in this document. This does not affect the existing options for extending HTTP, such as defining new methods, status codes, or fields.

Extensions are permitted to use new frame types (Section 7.2), new settings (Section 7.2.4.1), new error codes (Section 8), or new unidirectional stream types (Section 6.2). Registries are established for managing these extension points: frame types (Section 11.2.1), settings (Section 11.2.2), error codes (Section 11.2.3), and stream types (Section 11.2.4).

Implementations MUST ignore unknown or unsupported values in all extensible protocol elements. Implementations MUST discard frames and abort reading on unidirectional streams that have unknown or unsupported types. This means that any of these extension points can be safely used by extensions without prior arrangement or negotiation. However, where a known frame type is required to be in a specific location, such as the SETTINGS frame as the first frame of the control stream (see Section 6.2.1), an unknown frame type does not satisfy that requirement and SHOULD be treated as an error.



Extensions that could change the semantics of existing protocol components **MUST** be negotiated before being used. For example, an extension that changes the layout of the HEADERS frame cannot be used until the peer has given a positive signal that this is acceptable. Coordinating when such a revised layout comes into effect could prove complex. As such, allocating new identifiers for new definitions of existing protocol elements is likely to be more effective.

This document does not mandate a specific method for negotiating the use of an extension but notes that a setting (Section 7.2.4.1) could be used for that purpose. If both peers set a value that indicates willingness to use the extension, then the extension can be used. If a setting is used for extension negotiation, the default value **MUST** be defined in such a fashion that the extension is disabled if the setting is omitted.

## 10. Security Considerations

The security considerations of HTTP/3 should be comparable to those of HTTP/2 with TLS. However, many of the considerations from Section 10 of [HTTP2] apply to [QUIC-TRANSPORT] and are discussed in that document.

### 10.1. Server Authority

HTTP/3 relies on the HTTP definition of authority. The security considerations of establishing authority are discussed in Section 17.1 of [SEMANTICS].

### 10.2. Cross-Protocol Attacks

The use of ALPN in the TLS and QUIC handshakes establishes the target application protocol before application-layer bytes are processed. This ensures that endpoints have strong assurances that peers are using the same protocol.

This does not guarantee protection from all cross-protocol attacks. Section 21.5 of [QUIC-TRANSPORT] describes some ways in which the plaintext of QUIC packets can be used to perform request forgery against endpoints that don't use authenticated transports.

### 10.3. Intermediary Encapsulation Attacks

The HTTP/3 field encoding allows the expression of names that are not valid field names in the syntax used by HTTP (Section 5.1 of [SEMANTICS]). Requests or responses containing invalid field names MUST be treated as malformed (Section 4.1.3). An intermediary therefore cannot translate an HTTP/3 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/3 can transport field values that are not valid. While most values that can be encoded will not alter field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a field value MUST be treated as malformed (Section 4.1.3). Valid characters are defined by the "field-content" ABNF rule in Section 5.5 of [SEMANTICS].

### 10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH\_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server MUST ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Clients are required to reject pushed responses for which an origin server is not authoritative; see Section 4.4.

### 10.5. Denial-of-Service Considerations

An HTTP/3 connection can demand a greater commitment of resources to operate than an HTTP/1.1 or HTTP/2 connection. The use of field compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH\_PROMISE frames is constrained in a similar fashion. A client that accepts server push SHOULD limit the number of Push IDs it issues at a time.

Processing capacity cannot be guarded as effectively as state capacity.

The ability to send undefined protocol elements that the peer is required to ignore can be abused to cause a peer to expend additional processing time. This might be done by setting multiple undefined SETTINGS parameters, unknown frame types, or unknown stream types. Note, however, that some uses are entirely legitimate, such as optional-to-understand extensions and padding to increase resistance to traffic analysis.

Compression of field sections also offers some opportunities to waste processing resources; see Section 7 of [QPACK] for more details on potential abuses.

All these features -- i.e., server push, unknown protocol elements, field compression -- have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that does not monitor such behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error of type H3\_EXCESSIVE\_LOAD (Section 8), but false positives will result in disrupting valid connections and requests.

#### 10.5.1. Limits on Field Section Size

A large field section (Section 4.1) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header section, which prevents streaming of the header section to its ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean that an endpoint likely needs to buffer the entire header section. Since there is no hard limit to the size of a field section, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the SETTINGS\_MAX\_FIELD\_SECTION\_SIZE (Section 4.1.1.3) setting to advise peers of limits that might apply on the size of field sections. This setting is only advisory, so endpoints MAY choose to send field sections that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to an HTTP/3 connection, so any request or

response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger field section than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code ([RFC6585]). A client can discard responses that it cannot process.

#### 10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on a proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. Therefore, a proxy that supports CONNECT might be more conservative in the number of simultaneous requests it accepts.

A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME\_WAIT state. To account for this, a proxy might delay increasing the QUIC stream limits for some time after a TCP connection terminates.

#### 10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/3 enables compression of fields (Section 4.1.1); the following concerns also apply to the use of HTTP compressed content-codings; see Section 8.4.1 of [SEMANTICS].

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression contexts are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined.

Further considerations regarding the compression of field sections are described in [QPACK].

### 10.7. Padding and Traffic Analysis

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Where HTTP/2 employs PADDING frames and Padding fields in other frames to make a connection more resistant to traffic analysis, HTTP/3 can either rely on transport-layer padding or employ the reserved frame and stream types discussed in Section 7.2.8 and Section 6.2.3. These methods of padding produce different results in terms of the granularity of padding, how padding is arranged in relation to the information that is being protected, whether padding is applied in the case of packet loss, and how an implementation might control padding.

Reserved stream types can be used to give the appearance of sending traffic even when the connection is idle. Because HTTP traffic often occurs in bursts, apparent traffic can be used to obscure the timing or duration of such bursts, even to the point of appearing to send a constant stream of data. However, as such traffic is still flow controlled by the receiver, a failure to promptly drain such streams and provide additional flow control credit can limit the sender's ability to send real traffic.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Use of padding can result in less protection than might seem immediately obvious. Redundant padding could even be counterproductive. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

### 10.8. Frame Parsing

Several protocol elements contain nested length elements, typically in the form of frames with an explicit length containing variable-length integers. This could pose a security risk to an incautious implementer. An implementation MUST ensure that the length of a frame exactly matches the length of the fields it contains.

### 10.9. Early Data

The use of 0-RTT with HTTP/3 creates an exposure to replay attack. The anti-replay mitigations in [HTTP-REPLAY] MUST be applied when using HTTP/3 with 0-RTT. When applying [HTTP-REPLAY] to HTTP/3, references to the TLS layer refer to the handshake performed within QUIC, while all references to application data refer to the contents of streams.

### 10.10. Migration

Certain HTTP implementations use the client address for logging or access-control purposes. Since a QUIC client's address might change during a connection (and future versions might support simultaneous use of multiple addresses), such implementations will need to either actively retrieve the client's current address or addresses when they are relevant or explicitly accept that the original address might change.

### 10.11. Privacy Considerations

Several characteristics of HTTP/3 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client.

HTTP/3's preference for using a single QUIC connection allows correlation of a user's activity on a site. Reusing connections for different origins allows for correlation of activity across those origins.

Several features of QUIC solicit immediate responses and can be used by an endpoint to measure latency to their peer; this might have privacy implications in certain scenarios.

## 11. IANA Considerations

This document registers a new ALPN protocol ID (Section 11.1) and creates new registries that manage the assignment of codepoints in HTTP/3.

### 11.1. Registration of HTTP/3 Identification String

This document creates a new registration for the identification of HTTP/3 in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "h3" string identifies HTTP/3:

Protocol: HTTP/3

Identification Sequence: 0x68 0x33 ("h3")

Specification: This document

### 11.2. New Registries

New registries created in this document operate under the QUIC registration policy documented in Section 22.1 of [QUIC-TRANSPORT]. These registries all include the common set of fields listed in Section 22.1.1 of [QUIC-TRANSPORT]. These registries [SHALL be/are] collected under a "Hypertext Transfer Protocol version 3 (HTTP/3) Parameters" heading.

The initial allocations in these registries created in this document are all assigned permanent status and list a change controller of the IETF and a contact of the HTTP working group (ietf-http-wg@w3.org).

#### 11.2.1. Frame Types

This document establishes a registry for HTTP/3 frame type codes. The "HTTP/3 Frame Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [HTTP2], it is preferable that the assignments parallel each other where the code spaces overlap. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following field:

Frame Type: A name or label for the frame type.

Specifications of frame types MUST include a description of the frame layout and its semantics, including any parts of the frame that are conditionally present.

The entries in Table 2 are registered by this document.

Frame Type	Value	Specification
DATA	0x0	Section 7.2.1
HEADERS	0x1	Section 7.2.2
Reserved	0x2	N/A
CANCEL_PUSH	0x3	Section 7.2.3
SETTINGS	0x4	Section 7.2.4
PUSH_PROMISE	0x5	Section 7.2.5
Reserved	0x6	N/A
GOAWAY	0x7	Section 7.2.6
Reserved	0x8	N/A
Reserved	0x9	N/A
MAX_PUSH_ID	0xd	Section 7.2.7

Table 2: Initial HTTP/3 Frame Types

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.



## 11.2.2. Settings Parameters

This document establishes a registry for HTTP/3 settings. The "HTTP/3 Settings" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

While this registry is separate from the "HTTP/2 Settings" registry defined in [HTTP2], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation. Expert reviewers MAY reject unrelated registrations which would conflict with the same value in the corresponding registry.

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

**Setting Name:** A symbolic name for the setting. Specifying a setting name is optional.

**Default:** The value of the setting unless otherwise indicated. A default SHOULD be the most restrictive possible value.

The entries in Table 3 are registered by this document.

Setting Name	Value	Specification	Default
Reserved	0x0	N/A	N/A
Reserved	0x2	N/A	N/A
Reserved	0x3	N/A	N/A
Reserved	0x4	N/A	N/A
Reserved	0x5	N/A	N/A
MAX_FIELD_SECTION_SIZE	0x6	Section 7.2.4.1	Unlimited

Table 3: Initial HTTP/3 Settings

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

### 11.2.3. Error Codes

This document establishes a registry for HTTP/3 error codes. The "HTTP/3 Error Code" registry manages a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated. Use of values that are registered in the "HTTP/2 Error Code" registry is discouraged, and expert reviewers MAY reject such registrations.

In addition to common fields as described in Section 11.2, this registry includes two additional fields. Permanent registrations in this registry MUST include the following field:

Name: A name for the error code.

Description: A brief description of the error code semantics.

The entries in Table 4 are registered by this document. These error codes were selected from the range that operates on a Specification Required policy to avoid collisions with HTTP/2 error codes.

Name	Value	Description	Specification
H3_NO_ERROR	0x100	No error	Section 8.1
H3_GENERAL_PROTOCOL_ERROR	0x101	General protocol error	Section 8.1
H3_INTERNAL_ERROR	0x102	Internal error	Section 8.1
H3_STREAM_CREATION_ERROR	0x103	Stream	Section 8.1

		creation error	
H3_CLOSED_CRITICAL_STREAM	0x104	Critical stream was closed	Section 8.1
H3_FRAME_UNEXPECTED	0x105	Frame not permitted in the current state	Section 8.1
H3_FRAME_ERROR	0x106	Frame violated layout or size rules	Section 8.1
H3_EXCESSIVE_LOAD	0x107	Peer generating excessive load	Section 8.1
H3_ID_ERROR	0x108	An identifier was used incorrectly	Section 8.1
H3_SETTINGS_ERROR	0x109	SETTINGS frame contained invalid values	Section 8.1
H3_MISSING_SETTINGS	0x10a	No SETTINGS frame received	Section 8.1
H3_REQUEST_REJECTED	0x10b	Request not processed	Section 8.1
H3_REQUEST_CANCELLED	0x10c	Data no longer needed	Section 8.1
H3_REQUEST_INCOMPLETE	0x10d	Stream terminated early	Section 8.1

H3_MESSAGE_ERROR	0x10e	Malformed message	Section 8.1
H3_CONNECT_ERROR	0x10f	TCP reset or error on CONNECT request	Section 8.1
H3_VERSION_FALLBACK	0x110	Retry over HTTP/1.1	Section 8.1

Table 4: Initial HTTP/3 Error Codes

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

#### 11.2.4. Stream Types

This document establishes a registry for HTTP/3 unidirectional stream types. The "HTTP/3 Stream Type" registry governs a 62-bit space. This registry follows the QUIC registry policy; see Section 11.2. Permanent registrations in this registry are assigned using the Specification Required policy ([RFC8126]), except for values between 0x00 and 0x3f (in hexadecimal; inclusive), which are assigned using Standards Action or IESG Approval as defined in Sections 4.9 and 4.10 of [RFC8126].

In addition to common fields as described in Section 11.2, permanent registrations in this registry MUST include the following fields:

**Stream Type:** A name or label for the stream type.

**Sender:** Which endpoint on an HTTP/3 connection may initiate a stream of this type. Values are "Client", "Server", or "Both".

Specifications for permanent registrations MUST include a description of the stream type, including the layout and semantics of the stream contents.

The entries in the following table are registered by this document.

Stream Type	Value	Specification	Sender
Control Stream	0x00	Section 6.2.1	Both
Push Stream	0x01	Section 4.4	Server

Table 5

Each code of the format "0x1f \* N + 0x21" for non-negative integer values of N (that is, 0x21, 0x40, ..., through 0x3fffffffffffffffe) MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

## 12. References

### 12.1. Normative References

- [ALTSVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <<https://www.rfc-editor.org/info/rfc7838>>.
- [CACHING] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-14.txt>>.
- [HTTP-REPLAY] Thomson, M., Nottingham, M., and W. Tarreau, "Using Early Data in HTTP", RFC 8470, DOI 10.17487/RFC8470, September 2018, <<https://www.rfc-editor.org/info/rfc8470>>.
- [QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-21, 2 February 2021, <<https://tools.ietf.org/html/draft-ietf-quic-qpack-21>>.
- [QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-34, 2 February 2021, <<https://tools.ietf.org/html/draft-ietf-quic-transport-34>>.
- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<https://www.rfc-editor.org/info/rfc793>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [SEMANTICS] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-semantics-14.txt>>.
- [URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.

## 12.2. Informative References

- [BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <<http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>>.

## [DNS-TERMS]

Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <<https://www.rfc-editor.org/info/rfc8499>>.

## [HPACK]

Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<https://www.rfc-editor.org/info/rfc7541>>.

## [HTTP11]

Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-14, 12 January 2021, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-messaging-14.txt>>.

## [HTTP2]

Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.

## [RFC6585]

Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <<https://www.rfc-editor.org/info/rfc6585>>.

## [RFC8164]

Nottingham, M. and M. Thomson, "Opportunistic Security for HTTP/2", RFC 8164, DOI 10.17487/RFC8164, May 2017, <<https://www.rfc-editor.org/info/rfc8164>>.

## [TFO]

Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

## [TLS13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

## Appendix A. Considerations for Transitioning from HTTP/2

HTTP/3 is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/3, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/3.

HTTP/3 begins from the premise that similarity to HTTP/2 is preferable, but not a hard requirement. HTTP/3 departs from HTTP/2 where QUIC differs from TCP, either to take advantage of QUIC features (like streams) or to accommodate important shortcomings (such as a lack of total ordering). These differences make HTTP/3 similar to HTTP/2 in key aspects, such as the relationship of requests and responses to streams. However, the details of the HTTP/3 design are substantially different from HTTP/2.

Some important departures are noted in this section.

#### A.1. Streams

HTTP/3 permits use of a larger number of streams ( $2^{62}-1$ ) than HTTP/2. The same considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

In contrast to HTTP/2, stream concurrency in HTTP/3 is managed by QUIC. QUIC considers a stream closed when all data has been received and sent data has been acknowledged by the peer. HTTP/2 considers a stream closed when the frame containing the END\_STREAM bit has been committed to the transport. As a result, the stream for an equivalent exchange could remain "active" for a longer period of time. HTTP/3 servers might choose to permit a larger number of concurrent client-initiated bidirectional streams to achieve equivalent concurrency to HTTP/2, depending on the expected usage patterns.

In HTTP/2, only request and response bodies (the frame payload of DATA frames) are subject to flow control. All HTTP/3 frames are sent on QUIC streams, so all frames on all streams are flow-controlled in HTTP/3.

Due to the presence of other unidirectional stream types, HTTP/3 does not rely exclusively on the number of concurrent unidirectional streams to control the number of concurrent in-flight pushes. Instead, HTTP/3 clients use the MAX\_PUSH\_ID frame to control the number of pushes received from an HTTP/3 server.



## A.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END\_STREAM flag is not required. This permits the removal of the Flags field from the generic frame layout.

Frame payloads are largely drawn from [HTTP2]. However, QUIC includes many features (e.g., flow control) that are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/3. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/3 implementations. However, even frame types that appear in both mappings do not have identical semantics.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame type makes assumptions that frames from different streams will still be received in the order sent, HTTP/3 will break them.

Some examples of feature adaptations are described below, as well as general guidance to extension frame implementors converting an HTTP/2 extension to HTTP/3.

### A.2.1. Prioritization Differences

HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. HTTP/3 does not provide a means of signaling priority.

Note that while there is no explicit signaling for priority, this does not mean that prioritization is not important for achieving good performance.

### A.2.2. Field Compression Differences

HPACK was designed with the assumption of in-order delivery. A sequence of encoded field sections must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync.

Because this total ordering is not provided by QUIC, HTTP/3 uses a modified version of HPACK, called QPACK. QPACK uses a single unidirectional stream to make all modifications to the dynamic table, ensuring a total order of updates. All frames that contain encoded fields merely reference the table state at a given time without modifying it.

[QPACK] provides additional details.

#### A.2.3. Flow Control Differences

HTTP/2 specifies a stream flow control mechanism. Although all HTTP/2 frames are delivered on streams, only the DATA frame payload is subject to flow control. QUIC provides flow control for stream data and all HTTP/3 frame types defined in this document are sent on streams. Therefore, all frame headers and payload are subject to flow control.

#### A.2.4. Guidance for New Frame Type Definitions

Frame type definitions in HTTP/3 often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allows for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/3 use an identifier other than a Stream ID (e.g., Push IDs). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Because the Flags field is not present in generic HTTP/3 frames, those frames that depend on the presence of flags need to allocate space for flags as part of their frame payload.

Other than these issues, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with a control stream in HTTP/3. HTTP/3 extensions will not assume ordering, but would not be harmed by ordering, and are expected to be portable to HTTP/2.

#### A.2.5. Comparison Between HTTP/2 and HTTP/3 Frame Types

DATA (0x0): Padding is not defined in HTTP/3 frames. See Section 7.2.1.

HEADERS (0x1): The PRIORITY region of HEADERS is not defined in HTTP/3 frames. Padding is not defined in HTTP/3 frames. See Section 7.2.2.

PRIORITY (0x2): As described in Appendix A.2.1, HTTP/3 does not

provide a means of signaling priority.

**RST\_STREAM (0x3):** RST\_STREAM frames do not exist in HTTP/3, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL\_PUSH frame (Section 7.2.3).

**SETTINGS (0x4):** SETTINGS frames are sent only at the beginning of the connection. See Section 7.2.4 and Appendix A.3.

**PUSH\_PROMISE (0x5):** The PUSH\_PROMISE frame does not reference a stream; instead the push stream references the PUSH\_PROMISE frame using a Push ID. See Section 7.2.5.

**PING (0x6):** PING frames do not exist in HTTP/3, as QUIC provides equivalent functionality.

**GOAWAY (0x7):** GOAWAY does not contain an error code. In the client to server direction, it carries a Push ID instead of a server initiated stream ID. See Section 7.2.6.

**WINDOW\_UPDATE (0x8):** WINDOW\_UPDATE frames do not exist in HTTP/3, since QUIC provides flow control.

**CONTINUATION (0x9):** CONTINUATION frames do not exist in HTTP/3; instead, larger HEADERS/PUSH\_PROMISE frames than HTTP/2 are permitted.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/3 if still applicable. The IDs of frames defined in [HTTP2] have been reserved for simplicity. Note that the frame type space in HTTP/3 is substantially larger (62 bits versus 8 bits), so many HTTP/3 frame types have no equivalent HTTP/2 code points. See Section 11.2.1.

### A.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, as the first frame of the control stream, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/3. The HTTP-level setting that is retained in HTTP/3 has the same value as in HTTP/2. The superseded settings are reserved, and their receipt is an error. See Section 7.2.4.1 for discussion of both the retained and reserved values.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS\_HEADER\_TABLE\_SIZE (0x1): See [QPACK].

SETTINGS\_ENABLE\_PUSH (0x2): This is removed in favor of the MAX\_PUSH\_ID frame, which provides a more granular control over server push. Specifying a setting with the identifier 0x2 (corresponding to the SETTINGS\_ENABLE\_PUSH parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_MAX\_CONCURRENT\_STREAMS (0x3): QUIC controls the largest open Stream ID as part of its flow control logic. Specifying a setting with the identifier 0x3 (corresponding to the SETTINGS\_MAX\_CONCURRENT\_STREAMS parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_INITIAL\_WINDOW\_SIZE (0x4): QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying a setting with the identifier 0x4 (corresponding to the SETTINGS\_INITIAL\_WINDOW\_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_MAX\_FRAME\_SIZE (0x5): This setting has no equivalent in HTTP/3. Specifying a setting with the identifier 0x5 (corresponding to the SETTINGS\_MAX\_FRAME\_SIZE parameter) in the HTTP/3 SETTINGS frame is an error.

SETTINGS\_MAX\_HEADER\_LIST\_SIZE (0x6): This setting identifier has been renamed SETTINGS\_MAX\_FIELD\_SECTION\_SIZE.

In HTTP/3, setting values are variable-length integers (6, 14, 30, or 62 bits long) rather than fixed-length 32-bit fields as in HTTP/2. This will often produce a shorter encoding, but can produce a longer encoding for settings that use the full 32-bit space. Settings ported from HTTP/2 might choose to redefine their value to limit it to 30 bits for more efficient encoding, or to make use of the 62-bit space if more than 30 bits are required.

Settings need to be defined separately for HTTP/2 and HTTP/3. The IDs of settings defined in [HTTP2] have been reserved for simplicity. Note that the settings identifier space in HTTP/3 is substantially larger (62 bits versus 16 bits), so many HTTP/3 settings have no equivalent HTTP/2 code point. See Section 11.2.2.

As QUIC streams might arrive out of order, endpoints are advised not to wait for the peers' settings to arrive before responding to other streams. See Section 7.2.4.2.

#### A.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, the differences between HTTP/2 and HTTP/3 mean that error codes are not directly portable between versions.

The HTTP/2 error codes defined in Section 7 of [HTTP2] logically map to the HTTP/3 error codes as follows:

NO\_ERROR (0x0): H3\_NO\_ERROR in Section 8.1.

PROTOCOL\_ERROR (0x1): This is mapped to H3\_GENERAL\_PROTOCOL\_ERROR except in cases where more specific error codes have been defined. Such cases include H3\_FRAME\_UNEXPECTED, H3\_MESSAGE\_ERROR, and H3\_CLOSED\_CRITICAL\_STREAM defined in Section 8.1.

INTERNAL\_ERROR (0x2): H3\_INTERNAL\_ERROR in Section 8.1.

FLOW\_CONTROL\_ERROR (0x3): Not applicable, since QUIC handles flow control.

SETTINGS\_TIMEOUT (0x4): Not applicable, since no acknowledgment of SETTINGS is defined.

STREAM\_CLOSED (0x5): Not applicable, since QUIC handles stream management.

FRAME\_SIZE\_ERROR (0x6): H3\_FRAME\_ERROR error code defined in Section 8.1.

REFUSED\_STREAM (0x7): H3\_REQUEST\_REJECTED (in Section 8.1) is used to indicate that a request was not processed. Otherwise, not applicable because QUIC handles stream management.

CANCEL (0x8): H3\_REQUEST\_CANCELLED in Section 8.1.

COMPRESSION\_ERROR (0x9): Multiple error codes are defined in [QPACK].

CONNECT\_ERROR (0xa): H3\_CONNECT\_ERROR in Section 8.1.

ENHANCE\_YOUR\_CALM (0xb): H3\_EXCESSIVE\_LOAD in Section 8.1.

INADEQUATE\_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP\_1\_1\_REQUIRED (0xd): H3\_VERSION\_FALLBACK in Section 8.1.

Error codes need to be defined for HTTP/2 and HTTP/3 separately. See Section 11.2.3.

#### A.4.1. Mapping Between HTTP/2 and HTTP/3 Errors

An intermediary that converts between HTTP/2 and HTTP/3 may encounter error conditions from either upstream. It is useful to communicate the occurrence of error to the downstream but error codes largely reflect connection-local problems that generally do not make sense to propagate.

An intermediary that encounters an error from an upstream origin can indicate this by sending an HTTP status code such as 502, which is suitable for a broad class of errors.

There are some rare cases where it is beneficial to propagate the error by mapping it to the closest matching error type to the receiver. For example, an intermediary that receives an HTTP/2 stream error of type `REFUSED_STREAM` from the origin has a clear signal that the request was not processed and that the request is safe to retry. Propagating this error condition to the client as an HTTP/3 stream error of type `H3_REQUEST_REJECTED` allows the client to take the action it deems most appropriate. In the reverse direction, the intermediary might deem it beneficial to pass on client request cancellations that are indicated by terminating a stream with `H3_REQUEST_CANCELLED`; see Section 4.1.2.

Conversion between errors is described in the logical mapping. The error codes are defined in non-overlapping spaces in order to protect against accidental conversion that could result in the use of inappropriate or unknown error codes for the target version. An intermediary is permitted to promote stream errors to connection errors but they should be aware of the cost to the HTTP/3 connection for what might be a temporary or intermittent error.

#### Appendix B. Change Log

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

##### B.1. Since draft-ietf-quic-http-32

- \* Removed draft version guidance; added final version string
- \* Added `H3_MESSAGE_ERROR` for malformed messages

## B.2. Since draft-ietf-quic-http-31

Editorial changes only.

## B.3. Since draft-ietf-quic-http-30

Editorial changes only.

## B.4. Since draft-ietf-quic-http-29

- \* Require a connection error if a reserved frame type that corresponds to a frame in HTTP/2 is received (#3991, #3993)
- \* Require a connection error if a reserved setting that corresponds to a setting in HTTP/2 is received (#3954, #3955)

## B.5. Since draft-ietf-quic-http-28

- \* CANCEL\_PUSH is recommended even when the stream is reset (#3698, #3700)
- \* Use H3\_ID\_ERROR when GOAWAY contains a larger identifier (#3631, #3634)

## B.6. Since draft-ietf-quic-http-27

- \* Updated text to refer to latest HTTP revisions
- \* Use the HTTP definition of authority for establishing and coalescing connections (#253, #2223, #3558)
- \* Define use of GOAWAY from both endpoints (#2632, #3129)
- \* Require either :authority or Host if the URI scheme has a mandatory authority component (#3408, #3475)

## B.7. Since draft-ietf-quic-http-26

- \* No changes

## B.8. Since draft-ietf-quic-http-25

- \* Require QUICv1 for HTTP/3 (#3117, #3323)
- \* Remove DUPLICATE\_PUSH and allow duplicate PUSH\_PROMISE (#3275, #3309)
- \* Clarify the definition of "malformed" (#3352, #3345)

## B.9. Since draft-ietf-quic-http-24

- \* Removed H3\_EARLY\_RESPONSE error code; H3\_NO\_ERROR is recommended instead (#3130,#3208)
- \* Unknown error codes are equivalent to H3\_NO\_ERROR (#3276,#3331)
- \* Some error codes are reserved for greasing (#3325,#3360)

## B.10. Since draft-ietf-quic-http-23

- \* Removed "quic" Alt-Svc parameter (#3061,#3118)
- \* Clients need not persist unknown settings for use in 0-RTT (#3110,#3113)
- \* Clarify error cases around CANCEL\_PUSH (#2819,#3083)

## B.11. Since draft-ietf-quic-http-22

- \* Removed priority signaling (#2922,#2924)
- \* Further changes to error codes (#2662,#2551):
  - Error codes renumbered
  - HTTP\_MALFORMED\_FRAME replaced by HTTP\_FRAME\_ERROR, HTTP\_ID\_ERROR, and others
- \* Clarify how unknown frame types interact with required frame sequence (#2867,#2858)
- \* Describe interactions with the transport in terms of defined interface terms (#2857,#2805)
- \* Require the use of the "http-opportunistic" resource (RFC 8164) when scheme is "http" (#2439,#2973)
- \* Settings identifiers cannot be duplicated (#2979)
- \* Changes to SETTINGS frames in 0-RTT (#2972,#2790,#2945):
  - Servers must send all settings with non-default values in their SETTINGS frame, even when resuming
  - If a client doesn't have settings associated with a 0-RTT ticket, it uses the defaults



- Servers can't accept early data if they cannot recover the settings the client will have remembered
- \* Clarify that Upgrade and the 101 status code are prohibited (#2898, #2889)
- \* Clarify that frame types reserved for greasing can occur on any stream, but frame types reserved due to HTTP/2 correspondence are prohibited (#2997, #2692, #2693)
- \* Unknown error codes cannot be treated as errors (#2998, #2816)

B.12. Since draft-ietf-quic-http-21

No changes

B.13. Since draft-ietf-quic-http-20

- \* Prohibit closing the control stream (#2509, #2666)
- \* Change default priority to use an orphan node (#2502, #2690)
- \* Exclusive priorities are restored (#2754, #2781)
- \* Restrict use of frames when using CONNECT (#2229, #2702)
- \* Close and maybe reset streams if a connection error occurs for CONNECT (#2228, #2703)
- \* Encourage provision of sufficient unidirectional streams for QPACK (#2100, #2529, #2762)
- \* Allow extensions to use server-initiated bidirectional streams (#2711, #2773)
- \* Clarify use of maximum header list size setting (#2516, #2774)
- \* Extensive changes to error codes and conditions of their sending
  - Require connection errors for more error conditions (#2511, #2510)
  - Updated the error codes for illegal GOAWAY frames (#2714, #2707)
  - Specified error code for HEADERS on control stream (#2708)
  - Specified error code for servers receiving PUSH\_PROMISE (#2709)

- Specified error code for receiving DATA before HEADERS (#2715)
- Describe malformed messages and their handling (#2410, #2764)
- Remove HTTP\_PUSH\_ALREADY\_IN\_CACHE error (#2812, #2813)
- Refactor Push ID related errors (#2818, #2820)
- Rationalize HTTP/3 stream creation errors (#2821, #2822)

B.14. Since draft-ietf-quic-http-19

- \* SETTINGS\_NUM\_PLACEHOLDERS is 0x9 (#2443, #2530)
- \* Non-zero bits in the Empty field of the PRIORITY frame MAY be treated as an error (#2501)

B.15. Since draft-ietf-quic-http-18

- \* Resetting streams following a GOAWAY is recommended, but not required (#2256, #2457)
- \* Use variable-length integers throughout (#2437, #2233, #2253, #2275)
  - Variable-length frame types, stream types, and settings identifiers
  - Renumbered stream type assignments
  - Modified associated reserved values
- \* Frame layout switched from Length-Type-Value to Type-Length-Value (#2395, #2235)
- \* Specified error code for servers receiving DUPLICATE\_PUSH (#2497)
- \* Use connection error for invalid PRIORITY (#2507, #2508)

B.16. Since draft-ietf-quic-http-17

- \* HTTP\_REQUEST\_REJECTED is used to indicate a request can be retried (#2106, #2325)
- \* Changed error code for GOAWAY on the wrong stream (#2231, #2343)

## B.17. Since draft-ietf-quic-http-16

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

## B.18. Since draft-ietf-quic-http-15

Substantial editorial reorganization; no technical changes.

## B.19. Since draft-ietf-quic-http-14

- \* Recommend sensible values for QUIC transport parameters (#1720, #1806)
- \* Define error for missing SETTINGS frame (#1697, #1808)
- \* Setting values are variable-length integers (#1556, #1807) and do not have separate maximum values (#1820)
- \* Expanded discussion of connection closure (#1599, #1717, #1712)
- \* HTTP\_VERSION\_FALLBACK falls back to HTTP/1.1 (#1677, #1685)

## B.20. Since draft-ietf-quic-http-13

- \* Reserved some frame types for grease (#1333, #1446)

- \* Unknown unidirectional stream types are tolerated, not errors; some reserved for grease (#1490, #1525)
- \* Require settings to be remembered for 0-RTT, prohibit reductions (#1541, #1641)
- \* Specify behavior for truncated requests (#1596, #1643)

B.21. Since draft-ietf-quic-http-12

- \* TLS SNI extension isn't mandatory if an alternative method is used (#1459, #1462, #1466)
- \* Removed flags from HTTP/3 frames (#1388, #1398)
- \* Reserved frame types and settings for use in preserving extensibility (#1333, #1446)
- \* Added general error code (#1391, #1397)
- \* Unidirectional streams carry a type byte and are extensible (#910, #1359)
- \* Priority mechanism now uses explicit placeholders to enable persistent structure in the tree (#441, #1421, #1422)

B.22. Since draft-ietf-quic-http-11

- \* Moved QPACK table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)

B.23. Since draft-ietf-quic-http-10

- \* Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.24. Since draft-ietf-quic-http-09

- \* Selected QCRAM for header compression (#228, #1117)
- \* The server\_name TLS extension is now mandatory (#296, #495)
- \* Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.25. Since draft-ietf-quic-http-08

- \* Clarified connection coalescing rules (#940, #1024)

B.26. Since draft-ietf-quic-http-07

- \* Changes for integer encodings in QUIC (#595, #905)
- \* Use unidirectional streams as appropriate (#515, #240, #281, #886)
- \* Improvement to the description of GOAWAY (#604, #898)
- \* Improve description of server push usage (#947, #950, #957)

B.27. Since draft-ietf-quic-http-06

- \* Track changes in QUIC error code usage (#485)

B.28. Since draft-ietf-quic-http-05

- \* Made push ID sequential, add MAX\_PUSH\_ID, remove SETTINGS\_ENABLE\_PUSH (#709)
- \* Guidance about keep-alive and QUIC PINGs (#729)
- \* Expanded text on GOAWAY and cancellation (#757)

B.29. Since draft-ietf-quic-http-04

- \* Cite RFC 5234 (#404)
- \* Return to a single stream per request (#245, #557)
- \* Use separate frame type and settings registries from HTTP/2 (#81)
- \* SETTINGS\_ENABLE\_PUSH instead of SETTINGS\_DISABLE\_PUSH (#477)
- \* Restored GOAWAY (#696)
- \* Identify server push using Push ID rather than a stream ID (#702, #281)
- \* DATA frames cannot be empty (#700)

B.30. Since draft-ietf-quic-http-03

None.

B.31. Since draft-ietf-quic-http-02

- \* Track changes in transport draft

B.32. Since draft-ietf-quic-http-01

- \* SETTINGS changes (#181):
  - SETTINGS can be sent only once at the start of a connection; no changes thereafter
  - SETTINGS\_ACK removed
  - Settings can only occur in the SETTINGS frame a single time
  - Boolean format updated
- \* Alt-Svc parameter changed from "v" to "quic"; format updated (#229)
- \* Closing the connection control stream or any message control stream is a fatal error (#176)
- \* HPACK Sequence counter can wrap (#173)
- \* 0-RTT guidance added
- \* Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.33. Since draft-ietf-quic-http-00

- \* Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)
- \* Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)
- \* Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01
- \* Reworked SETTINGS\_ACK to account for indeterminate inter-stream order (#75)
- \* Described CONNECT pseudo-method (#95)
- \* Updated ALPN token and Alt-Svc guidance (#13,#87)
- \* Application-layer-defined error codes (#19,#74)

B.34. Since draft-shade-quic-http2-mapping-00

- \* Adopted as base for draft-ietf-quic-http
- \* Updated authors/editors list

#### Acknowledgments

The original authors of this specification were Robbie Shade and Mike Warres.

The IETF QUIC Working Group received an enormous amount of support from many people. Among others, the following people provided substantial contributions to this document:

- \* Bence Beky
- \* Daan De Meyer
- \* Martin Duke
- \* Roy Fielding
- \* Alan Frindell
- \* Alessandro Ghedini
- \* Nick Harper
- \* Ryan Hamilton
- \* Christian Huitema
- \* Subodh Iyengar
- \* Robin Marx
- \* Patrick McManus
- \* Luca Niccolini
- \* (Kazuho Oku)
- \* Lucas Pardue
- \* Roberto Peon
- \* Julian Reschke

- \* Eric Rescorla
- \* Martin Seemann
- \* Ben Schwartz
- \* Ian Swett
- \* Willy Taureau
- \* Martin Thomson
- \* Dmitri Tikhonov
- \* Tatsuhiro Tsujikawa

A portion of Mike's contribution was supported by Microsoft during his employment there.

Author's Address

Mike Bishop (editor)  
Akamai

Email: [mbishop@evequefou.be](mailto:mbishop@evequefou.be)



Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: May 7, 2020

E. Kinnear  
T. Pauly  
Apple Inc.  
November 4, 2019

Using HTTP/2 as a Transport for Arbitrary Bytestreams  
draft-kinnear-httpbis-http2-transport-02

Abstract

HTTP/2 provides multiplexing of HTTP requests over a single underlying transport connection. HTTP/2 Transport defines the use of the bidirectional extended CONNECT handshake to negotiate the use of application protocols using streams of an HTTP/2 connection as transport.

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 7, 2020.

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 . . . . .	2
1.1. Notational Conventions . . . . .	2
2. The SETTINGS_ENABLE_BIDIRECTIONAL_CONNECT Parameter . . . . .	3
3. Negotiating Bidirectional Transport . . . . .	3
3.1. Initiating the Extended CONNECT Handshake . . . . .	4
3.2. Responding to the Extended CONNECT Handshake . . . . .	4
4. Using Tunnels Established via the Extended CONNECT Handshake . . . . .	5
4.1. Example . . . . .	5
5. IANA Considerations . . . . .	6
6. Security Considerations . . . . .	7
7. Acknowledgments . . . . .	7
8. Normative References . . . . .	7
Authors' Addresses . . . . .	8

## 1. Introduction

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 extends the CONNECT handshake to allow both endpoints of an HTTP/2 connection to establish streams that tunnel data. It also defines a protocol name for use in the extended CONNECT handshake that allow negotiation of HTTP/2 streams that transport arbitrary bytestreams. Being able to transport application protocol 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 Bidirectional Transport

[RFC6455] defines the pseudo-header field :protocol which can indicate the protocol intended to be used on the tunnel established by the CONNECT method. Values for the :protocol pseudo-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 new stream via the extended CONNECT method. Similarly, either endpoint may be required to respond to an incoming CONNECT request seeking to establish such a stream.

### 3.1. Initiating the Extended CONNECT Handshake

Endpoints using this mechanism to establish bidirectional transport over HTTP/2 streams follow the CONNECT handshake procedure defined in [RFC6455]. However, instead of supplying "websocket" for the :protocol psuedo-header field to indicate a WebSocket connection, they negotiate the use of a specific application protocol by specifying an appropriate value. This document registers "bytestream" as a value to be used when an out-of-band negotiation has already occurred and an application protocol wishes to transport arbitrary bytes on an HTTP/2 stream. Any endpoint supplying "bytestream" as a value for the :protocol psuedo-header MUST have previously negotiated the use of this value via another mechanism.

The :scheme and :path psuedo-headers are required by [RFC6455]. The scheme of the target URI MUST be set to "https" for all :protocol values. 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 to be that of the supplied :protocol value and follows the semantics defined by that protocol.

### 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 psuedo-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 to be of the supplied :protocol value.

#### 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 byte stream 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.

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.

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.

```
[[ From Client ]]
```

```
[[ From Server ]]
```

```
SETTINGS
```

```
SETTINGS_ENABLE_CONNECT[..] = 1
```

```
SETTINGS_ENABLE_BIDIRECTIONAL[..] = 1
```

```
SETTINGS
```

```
SETTINGS_ENABLE_CONNECT[..] = 1
```

```
SETTINGS_ENABLE_BIDIRECTIONAL[..] = 1
```

```
HEADERS + END_HEADERS
```

```
:method = CONNECT
```

```
:protocol = bytestream
```

```
:scheme = https
```

```
:path = /
```

```
:authority = server.example.com
```

```
HEADERS + END_HEADERS
```

```
:status = 200
```

```
DATA
```

```
Bytestream Data
```

```
DATA + END_STREAM
```

```
Bytestream Data
```

```
DATA + END_STREAM
```

```
Bytestream Data
```

## 5. IANA Considerations

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

A new token, "bytestream", for arbitrary bytestream data.

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

## 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, 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

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <<https://www.rfc-editor.org/info/rfc6455>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8441] McManus, P., "Bootstrapping WebSockets with HTTP/2", RFC 8441, DOI 10.17487/RFC8441, September 2018, <<https://www.rfc-editor.org/info/rfc8441>>.

Authors' Addresses

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

Email: [ekinnear@apple.com](mailto:ekinnear@apple.com)

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

Email: [tpauly@apple.com](mailto:tpauly@apple.com)



Individual submission  
Internet-Draft  
Intended status: Informational  
Expires: May 9, 2019

M. Kucherawy  
Facebook, Inc.  
November 5, 2018

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.]

- o Attacks that use dictionary-based compression to recover content in the response.
- o 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

not be exhaustive, but it is hopefully complete enough to ensure quality work is produced as a result.

## 6. References

### 6.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

### 6.2. Informative References

- [RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <<https://www.rfc-editor.org/info/rfc7932>>.
- [RFC8478] Collet, Y. and M. Kucherawy, Ed., "Zstandard Compression and the application/zstd Media Type", RFC 8478, DOI 10.17487/RFC8478, October 2018, <<https://www.rfc-editor.org/info/rfc8478>>.

## 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:

- o draft-lee-sdch-spec, which was implemented in Chrome but then withdrawn
- o draft-vkrasnov-h2-compression-dictionaries
- o draft-vandevenne-shared-brotli-format
- o HTTPBIS discussion during IETF 97
- o 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



Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: August 24, 2019

M. Nottingham  
Fastly  
P. Sikora  
Google  
February 20, 2019

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):

- o <https://docs.fastly.com/guides/debugging/common-503-errors>
- o <https://support.cloudflare.com/hc/en-us/sections/200820298-Error-Pages>
- o <https://cloud.google.com/load-balancing/docs/https/https-logging-monitoring>
- o <https://docs.google.com/document/d/1fMEK80KlpHcL4CwhniOupgQu4MDsxK4y4dKhthjjCMA>

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

3.17. HTTP Response Header Too Large . . . . .	10
3.18. HTTP Response Body Too Large . . . . .	10
3.19. HTTP Response Transfer-Coding Error . . . . .	11
3.20. HTTP Response Content-Coding Error . . . . .	11
3.21. HTTP Response Timeout . . . . .	11
3.22. TLS Handshake Error . . . . .	12
3.23. TLS Untrusted Peer Certificate . . . . .	12
3.24. TLS Expired Peer Certificate . . . . .	12
3.25. TLS Unexpected Peer Certificate . . . . .	12
3.26. TLS Unexpected Peer Identity . . . . .	13
3.27. TLS Missing Proxy Certificate . . . . .	13
3.28. TLS Rejected Proxy Certificate . . . . .	13
3.29. TLS Error . . . . .	13
3.30. HTTP Request Error . . . . .	14
3.31. HTTP Request Denied . . . . .	14
3.32. HTTP Upgrade Failed . . . . .	14
3.33. Proxy Internal Error . . . . .	14
4. Defining New Proxy Status Types . . . . .	15
5. IANA Considerations . . . . .	16
6. Security Considerations . . . . .	16
7. References . . . . .	16
7.1. Normative References . . . . .	16
7.2. URIs . . . . .	17
Authors' Addresses . . . . .	17

## 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 is 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.

- o proxy - a sh-token identifying the HTTP intermediary generating this response.
- o origin - a sh-token identifying the origin server whose behaviour triggered this response.
- o 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.
- o tries - a sh-integer indicating the number of times that the error has occurred before this response.

- o details - a sh-string containing additional information not captured anywhere else. This can include implementation-specific or deployment-specific information.

### 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

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

#### 3.2. DNS Error

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

#### 3.3. Destination Not Found

- o Name: destination\_not\_found
- o 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.
- o Extra Parameters: None.
- o 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

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

### 3.13. HTTP Response Status

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

### 3.14. HTTP Incomplete Response

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

### 3.15. HTTP Protocol Error

- o Name: `http_protocol_error`
- o 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.
- o 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.



- o Recommended HTTP status code: 502

### 3.16. HTTP Response Header Block Too Large

- o Name: `http_response_header_block_size`
- o Description: The intermediary received a response to the request whose header block was considered too large.
- o 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.
- o Recommended HTTP status code: 502

### 3.17. HTTP Response Header Too Large

- o Name: `http_response_header_size`
- o Description: The intermediary received a response to the request containing an individual header line that was considered too large.
- o Extra Parameters:
  - \* `header_name`: a sh-string indicating the name of the header that triggered the error.
- o Recommended HTTP status code: 502

### 3.18. HTTP Response Body Too Large

- o Name: `http_response_body_size`
- o Description: The intermediary received a response to the request whose body was considered too large.
- o 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.
- o Recommended HTTP status code: 502

### 3.19. HTTP Response Transfer-Coding Error

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

### 3.20. HTTP Response Content-Coding Error

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

### 3.21. HTTP Response Timeout

- o Name: `http_response_timeout`
- o Description: The intermediary reached a configured time limit waiting for the complete response.
- o Extra Parameters: None.
- o Recommended HTTP status code: 504

## 3.22. TLS Handshake Error

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

## 3.23. TLS Untrusted Peer Certificate

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

## 3.24. TLS Expired Peer Certificate

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

## 3.25. TLS Unexpected Peer Certificate

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

- o Recommended HTTP status code: 502

### 3.26. TLS Unexpected Peer Identity

- o Name: `tls_unexpected_peer_identity`
- o Description: The intermediary received peer certificate with unexpected identity (e.g., Subject Alternative Name doesn't match) during TLS handshake with the next hop.
- o Extra Parameters:
  - \* details: a sh-string containing the identity of the next hop.
- o Recommended HTTP status code: 502

### 3.27. TLS Missing Proxy Certificate

- o Name: `tls_missing_proxy_certificate`
- o Description: The next hop requested client certificate from the intermediary during TLS handshake, but it wasn't configured with one.
- o Extra Parameters: None.
- o Recommended HTTP status code: 500

### 3.28. TLS Rejected Proxy Certificate

- o Name: `tls_rejected_proxy_certificate`
- o Description: The next hop rejected client certificate provided by the intermediary during TLS handshake.
- o Extra Parameters: None.
- o Recommended HTTP status code: 500

### 3.29. TLS Error

- o Name: `tls_error`
- o Description: The intermediary encountered a TLS error when communicating with the next hop.
- o 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]

See the registry at <https://iana.org/assignments/http-proxy-statuses> [4] for details on where to send registration requests.

## 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]  
Nottingham, M. and P. Kamp, "Structured Headers for HTTP", draft-ietf-httpbis-header-structure-09 (work in progress), December 2018.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8499] Hoffman, P., Sullivan, A., and K. Fujiwara, "DNS Terminology", BCP 219, RFC 8499, DOI 10.17487/RFC8499, January 2019, <<https://www.rfc-editor.org/info/rfc8499>>.

## 7.2. URIs

- [1] <https://github.com/mnot/I-D/labels/proxy-status>
- [2] <https://mnot.github.io/I-D/proxy-status/>
- [3] <https://datatracker.ietf.org/doc/draft-nottingham-proxy-status/>
- [4] <https://iana.org/assignments/http-proxy-statuses>
- [5] <https://iana.org/assignments/http-proxy-statuses>

## Authors' Addresses

Mark Nottingham  
Fastly

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

Piotr Sikora  
Google

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



HTTPWG  
Internet-Draft  
Intended status: Best Current Practice  
Expires: March 22, 2020

B. Sniffen  
M. Bishop  
E. Nygren  
R. Salz  
Akamai Technologies  
September 19, 2019

Best practices for TLS Downgrade  
draft-richsalz-httpbis-https-downgrade-02

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."

This Internet-Draft will expire on March 22, 2020.

#### 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. Background and Motivation . . . . .	2
2. Recommended alternatives . . . . .	4
3. Potential risk mitigations . . . . .	4
4. Recommendations . . . . .	5
5. Alternative approaches . . . . .	5
6. Security Considerations . . . . .	6
6.1. Risks of doing downgrade . . . . .	6
6.2. Control of the network between the cache and the origin . . . . .	6
6.3. Confused-deputy issues at the browser or origin . . . . .	6
7. Normative References . . . . .	7
Appendix A. Acknowledgements . . . . .	7
Authors' Addresses . . . . .	7

#### 1. Background and Motivation

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

- o Show HTTP as "not secure"
- o Showing mixed-content warnings when images or advertisements are HTTP on an HTTPS base page
- o 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:

- o 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.
- o 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.
- o 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.
- o 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.
- o 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.
- o 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 downgrade

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.

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

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <<https://www.rfc-editor.org/info/rfc7258>>.

## 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](mailto:bsniffen@akamai.com)

Mike Bishop  
Akamai Technologies  
145 Broadway  
Cambridge 02139  
US

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

Erik Nygren  
Akamai Technologies  
145 Broadway  
Cambridge 02139  
US

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

Rich Salz  
Akamai Technologies  
145 Broadway  
Cambridge 02139  
US

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



Network Working Group  
Internet-Draft  
Intended status: Standards Track  
Expires: 28 January 2021

J. Yasskin  
Google  
27 July 2020

Signed HTTP Exchanges  
draft-yasskin-http-origin-signed-responses-09

## 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](mailto:ietf-http-wg@w3.org)), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/> (<https://lists.w3.org/Archives/Public/ietf-http-wg/>).

The source code and issues list for this draft can be found in <https://github.com/WICG/webpackage> (<https://github.com/WICG/webpackage>).

## 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 28 January 2021.

## Copyright Notice

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

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

## Table of Contents

1. Introduction . . . . .	4
2. Terminology . . . . .	4
3. Signing an exchange . . . . .	5
3.1. The Signature Header . . . . .	6
3.1.1. Examples . . . . .	7
3.1.2. Open Questions . . . . .	9
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 . . . . .	12
3.5.1. Open Questions . . . . .	16
3.6. Updating signature validity . . . . .	16
3.6.1. Examples . . . . .	17
3.7. The Accept-Signature header . . . . .	19
3.7.1. Integrity identifiers . . . . .	20
3.7.2. Key type identifiers . . . . .	20
3.7.3. Key value identifiers . . . . .	20
3.7.4. Examples . . . . .	21
3.7.5. Open Questions . . . . .	21
4. Cross-origin trust . . . . .	22
4.1. Uncached header fields . . . . .	23
4.1.1. Stateful header fields . . . . .	24
4.2. Certificate Requirements . . . . .	25
4.2.1. Extensions to the CAA Record: cansignhttpexchanges Parameter . . . . .	26
5. Transferring a signed exchange . . . . .	26
5.1. Same-origin response . . . . .	27
5.1.1. Serialized headers for a same-origin response . . . . .	27
5.1.2. The Signed-Headers Header . . . . .	28

5.2.	HTTP/2 extension for cross-origin Server Push . . . . .	28
5.2.1.	Indicating support for cross-origin Server Push . . . . .	28
5.2.2.	NO_TRUSTED_EXCHANGE_SIGNATURE error code . . . . .	29
5.2.3.	Validating a cross-origin Push . . . . .	29
5.3.	application/signed-exchange format . . . . .	30
5.3.1.	Cross-origin trust in application/signed-exchange . . . . .	31
5.3.2.	Example . . . . .	32
5.3.3.	Open Questions . . . . .	32
6.	Security considerations . . . . .	32
6.1.	Over-signing . . . . .	32
6.1.1.	Session fixation . . . . .	33
6.1.2.	Misleading content . . . . .	33
6.2.	Off-path attackers . . . . .	33
6.2.1.	Mis-issued certificates . . . . .	33
6.2.2.	Stolen private keys . . . . .	34
6.3.	Downgrades . . . . .	35
6.4.	Signing oracles are permanent . . . . .	35
6.5.	Unsigned headers . . . . .	35
6.6.	application/signed-exchange . . . . .	36
6.7.	Key re-use with TLS . . . . .	36
6.8.	Content sniffing . . . . .	36
7.	Privacy considerations . . . . .	37
7.1.	Visibility of resource requests . . . . .	38
7.2.	User ID transfer . . . . .	39
8.	IANA considerations . . . . .	39
8.1.	Signature Header Field Registration . . . . .	39
8.2.	Accept-Signature Header Field Registration . . . . .	39
8.3.	Signed-Headers Header Field Registration . . . . .	40
8.4.	HTTP/2 Settings . . . . .	40
8.5.	HTTP/2 Error code . . . . .	40
8.6.	Internet Media Type application/signed-exchange . . . . .	41
8.7.	Internet Media Type application/cert-chain+cbor . . . . .	42
8.8.	The cansignhttpexchanges CAA Parameter . . . . .	43
9.	References . . . . .	43
9.1.	Normative References . . . . .	43
9.2.	Informative References . . . . .	46
Appendix A.	Use cases . . . . .	49
A.1.	PUSHed subresources . . . . .	49
A.2.	Explicit use of a content distributor for subresources . . . . .	49
A.3.	Subresource Integrity . . . . .	50
A.4.	Binary Transparency . . . . .	50
A.5.	Static Analysis . . . . .	51
A.6.	Offline websites . . . . .	51
Appendix B.	Requirements . . . . .	51
B.1.	Proof of origin . . . . .	51
B.1.1.	Certificate constraints . . . . .	51
B.1.2.	Signature constraints . . . . .	52
B.1.3.	Retrieving the certificate . . . . .	52

B.2. How much to sign . . . . .	53
B.2.1. Conveying the signed headers . . . . .	53
B.3. Response lifespan . . . . .	54
B.3.1. Certificate revocation . . . . .	54
B.3.2. Response downgrade attacks . . . . .	54
B.4. Low implementation complexity . . . . .	55
B.4.1. Limited choices . . . . .	55
B.4.2. Bounded-buffering integrity checking . . . . .	55
Appendix C. Determining validity using cache control . . . . .	56
C.1. Example of updating cache control . . . . .	56
C.2. Downsides of updating cache control . . . . .	57
Appendix D. Change Log . . . . .	57
Appendix E. Acknowledgements . . . . .	60
Author's Address . . . . .	60

## 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-wpack-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 (<https://url.spec.whatwg.org/#concept-url-parser>) ([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 (<https://url.spec.whatwg.org/#absolute-url-string>) 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 ([http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1\\_chap04.html#tag\\_04\\_16](http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16)).

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 (however, this is likely to change soon; 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=*MEUCIQDXLI2gN3RNB1gFiuRNFpZXcDiaUpX6HIEwcZEc0cZYLAIGA9DsVOMM+g5YpwEBdGW3sS
+bvnmAJJiSMwhuBdqp5UY=*;
  integrity="digest/mi-sha256";
  validity-url="https://example.com/resource.validity.1511128380";
  cert-url="https://example.com/oldcerts";
  cert-sha256=*W7uB969dFW3Mb5ZefPS9Tq5ZbH5iSmOILpjv2qEArmI=*;
  date=1511128380; expires=1511733180,
sig2;
  sig=*MEQCIGjZRqTRf9iKNkGFyzRMTFgwf/BrY2ZNIP/dykhUV0aYAiBTXg+8wujoT4n/W+cNgb7pGq
QvIUgYZ8u8HZJ5YH26Qg==*;
  integrity="digest/mi-sha256";
  validity-url="https://example.com/resource.validity.1511128380";
  cert-url="https://example.com/newcerts";
  cert-sha256=*J/1Em9kNR0DdCmINbvitpvdYKNQ+YgBj99DlYp4fEXw=*;
  date=1511128380; expires=1511733180,
srisig;
  sig=*1GZVaJJM5f2oGczFlLmBdKTDL+QADza4BgeO494ggACYJOvrof6uh5OJCcwKrk7DK+LBch0jss
DYPp5CLc1SDA==*;
  integrity="digest/mi-sha256";
  validity-url="https://example.com/resource.validity.1511128380";
  ed25519key=*zsSevyFsxyZHiUluVBDD4eypdRLTqyWRVOJuuKUz+A8=*
  date=1511128380; expires=1511733180,
thirdpartysig;
  sig=*MEYCIQCnXJzn6Rh2fNxsobktir8TkiaJYQFhWTuWiIi4PewQaQIhAMS2TVjc4rTshDtXbgQEOW
gj2mRXALhfXPztXgPupii+*;
  integrity="digest/mi-sha256";
  validity-url="https://thirdparty.example.com/resource.validity.1511161860";
  cert-url="https://thirdparty.example.com/certs";
  cert-sha256=*UeOWUPkvxlGRTyvHcsMUN0A2oNsZbU8EUvg8A9ZAnNc=*;
  date=1511133060; expires=1511478660,

```

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

The next revision of [I-D.ietf-httpbis-header-structure] will provide a way to parameterise byte sequences, at which point the signature itself is likely to become the main list item.

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+pvQwPywfFvAulUeFrs=
Signed-Headers: "content-type", "digest"
```

```
<!doctype html>
<html>
...
```

The cbor representation consists of the following item, represented using the extended diagnostic notation from [CDDL] appendix G:

```
{
  'digest': 'mi-sha256=dcRDgR2GM35DluAV13PzgnG6+pvQwPywfFvAulUeFrs=',
  '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:

```
cert-chain = [
  "", ; U+1F4DC U+26D3
  + augmented-certificate
]
augmented-certificate = {
  cert: bytes,
  ? ocsf: bytes,
  ? sct: bytes,
  * tstr => any,
}
```

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 OCSF 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 bitwise 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. "z", 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-Type-Options: nosniff" header field ([FETCH]) to the extracted response.

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 OCSF 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 OCSF responses by themselves. Instead, clients need to re-fetch the "cert-url" (Section 3.5, Paragraph 6) to get a chain including a newer OCSF response.

The "validity-url" parameter (Section 3.1) 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 ([CDDL]):

```
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).

## 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:

```

{
  "signatures": [
    'sig1; '
    'sig=*MEQCIC/I9Q+7BZFP6cSDsWx43pBAL0ujTbON/+7RwKVk+ba5AiB3FSFLZqpzmDJ0NumNwN0
4pqqJZE99fcK86UjkPbj4jw==*; '
    'validity-url="https://example.com/resource.validity.1511157180"; '
    'integrity="digest/mi-sha256"; '
    'cert-url="https://example.com/newcerts"; '
    'cert-sha256=*J/lEm9kNRODdCmINbvitpvdYKNQ+YgBj99DlYp4fEXw=*; '
    'date=1511733180; expires=1512337980'
  ],
  "update": {
    "size": 5557452
  }
}

```

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 (<https://www.iana.org/assignments/http-dig-alg/http-dig-alg.xhtml>) registry named in lower-case 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 lower-case 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

"11qYAYKxCrfVS/7TyWQHOG7hcvPapiMlrwIaaPcHUro=", valid "Accept-Signature" header fields include:

```
Accept-Signature: ..., ed25519key; *11qYAYKxCrfVS/7TyWQHOG7hcvPapiMlrwIaaPcHUro=*
Accept-Signature: ..., ed25519key; *11qYAYKxCrfVS/7TyWQHOG==*
Accept-Signature: ..., ed25519key; *11qYAQ==*
Accept-Signature: ..., ed25519key; **
```

but not

```
Accept-Signature: ..., ed25519key; *11qYA===*
```

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='...'>" 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 (Section 3.1) is not same-origin (<https://html.spec.whatwg.org/multipage/origin.html#same-origin>) 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]
- \* "Proxy-Authenticate", [RFC7235]
- \* "Proxy-Authentication-Info", [RFC7615]
- \* "Public-Key-Pins", [RFC7469]
- \* "Sec-WebSocket-Accept", [RFC6455]
- \* "Set-Cookie", [RFC6265]



- \* "Set-Cookie2", [RFC2965]
- \* "SetProfile", [W3C.NOTE-OPS-OverHTTP]
- \* "Strict-Transport-Security", [RFC6797]
- \* "WWW-Authenticate", [RFC7235]

#### 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.

Certificates with this extension MUST have a Validity Period no greater than 90 days.

Conforming CAs MUST NOT mark this extension as critical.

A conforming CA MUST NOT issue certificates with this extension unless, for each `dnsName` in the `subjectAltName` extension of the certificate to be issued:

1. An "issue" or "issuwild" CAA property ([RFC6844]) exists that authorizes the CA to issue the certificate; and

2. The "cansignhttpexchanges" parameter (Section 4.2.1) is present on the property and is equal to "yes"

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.

Some certificates have already been issued with this extension and with validity periods longer than 90 days. These certificates will not immediately be treated as invalid. Instead:

- \* Clients MUST reject certificates with this extension that were issued after 2019-05-01 and have a Validity Period longer than 90 days.
- \* After 2019-08-01, clients MUST reject all certificates with this extension that have a Validity Period longer than 90 days.

The above requirements on CAs to limit the Validity Period and check for a CAA parameter are effective starting 2019-05-01.

#### 4.2.1. Extensions to the CAA Record: cansignhttpexchanges Parameter

A CAA parameter "cansignhttpexchanges" is defined for the "issue" and "issuewild" properties defined by [RFC6844]. The value of this parameter, if specified, MUST be "yes".

#### 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 "sxml-".

2. 2 bytes storing a big-endian integer "fallbackUrlLength".
3. "fallbackUrlLength" bytes holding a "fallbackUrl", which MUST UTF-8 decode to 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 ( $16 \times 1024$ ), parsing MUST fail.
5. 3 bytes storing a big-endian integer "headerLength". If this is larger than 524288 ( $512 \times 1024$ ), 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/` (`https://example.com/`) follows, with lengths represented by descriptions in "<>"s, CBOR represented in the extended diagnostic format defined in Appendix G of [CDDL], and most of the "Signature" header field and payload elided with a ....:

```
sxgl\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>sigl; sig=*...; integrity="digest/mi-sha256"; ...{
  'status': '200',
  'content-type': 'text/html'
}<!doctype html>\r\n<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 or IP routing, 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], and are discussed further in [I-D.bishop-httpbis-origin-fed-up].

#### 6.2.1. Mis-issued certificates

If a CA mis-issues a certificate for a domain, this specification provides a way to detect the mis-issuance and mitigate harm within approximately two weeks. Specifically, because all signed exchanges must include a "SignedCertificateTimestampList" ([RFC6962], a CT log has promised to publish the mis-issued certificate within that log's Maximum Merge Delay, 1 day for many logs. The domain owner can then detect the mis-issued certificate and notify the CA to revoke it,

which the [BRs], section 4.9.1.1, say they must do within another 5 days.

Once the mis-issued certificate is revoked, existing OCSP responses begin to expire. The [BRs], section 4.9.10, require that OCSP responses have a maximum expiration time of 10 days, after which they can't be used to validate a certificate chain (Section 3.3). This leads to a total compromised time of 16 days after a mis-issuance.

However, CAs might future-date their OCSP responses, in which case the mitigation doesn't work.

CAs are forbidden from future-dating their OCSP responses by the [BRs] section 4.9.9, "OCSP responses MUST conform to RFC6960 and/or RFC5019." [RFC6960] includes, "The time at which the status was known to be correct SHALL be reflected in the thisUpdate field of the response.", and [RFC5019] includes, "When pre-producing OCSPResponse messages, the responder MUST set the thisUpdate, nextUpdate, and producedAt times as follows: thisUpdate: The time at which the status being indicated is known to be correct."

However, if a CA violates the [BRs] to sign future-dated OCSP responses, attempts to keep the nonconformant OCSP responses private, but then leaks them, it could cause clients to trust a hostile signed exchange long after its certificate has been revoked.

Clients could use systems like [CRLSets] and [OneCrl] to revoke the intermediate certificate that signed the future-dated OCSP responses.

#### 6.2.2. Stolen private keys

If the private key for a CanSignHttpExchanges certificate is stolen, it can be used at scale until the certificate expires or is revoked, and unlike for a stolen key for a normal TLS-terminating certificate, the rightful owner can't detect the problem by watching for attacks on the DNS or routing infrastructure.

This specification does not currently propose a way for the rightful owner to detect that their keys are being used by an attacker, after they've opted into the risk by requesting a CanSignHttpExchanges certificate in the first place. Clients can fetch a signature's "validity-url" (Section 3.1) to help owners detect key compromise, but that compromises some of the privacy properties of this specification.

### 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" (Section 3.1) 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 OSCP 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 (<https://url.spec.whatwg.org/#percent-encode>) ([URL]) any bytes that are greater than 0x7E or are not URL code points (<https://url.spec.whatwg.org/#url-code-points>) ([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

### 7.1. Visibility of resource requests

Normally, when a client follows a link from `https://source.example/page.html` to `"https://publisher.example/page.html"`, `"publisher.example"` learns that the client is interested in the resource. `"source.example"` also has several ways of discovering that the client has clicked the link, including the use of Javascript to record the click or having the link point to a URL that serves a 302 redirect to the real target.

If `"publisher.example"` signs `"page.html"` into `"page.sxg"`, `"distributor.example"` serves it as `"https://distributor.example/publisher/page.sxg"`, and the client fetches it from there, then `"distributor.example"` learns that the client is interested, and if the client executes some Javascript on the page or makes subresource requests, that could also report the client's interest back to `"publisher.example"`.

To prevent network operators other than `"distributor.example"` or `"publisher.example"` 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.

If `"source.example"` and `"distributor.example"` are controlled by the same entity, no extra information escapes here. If they are run by different entities, a similar amount of information escapes as if `"source.example"` had implemented its click tracking by outsourcing to a service like `https://bit.ly/` (`https://bit.ly/`).

There has been discussion of allowing a publisher to restrict the set of distributors that can host its signed content. If that's added, then the privacy situation becomes more similar to the situation with CDNs, where a publisher chooses a CDN to serve their content, and the CDN learns about all requests for that content. Here the publisher would choose one or more distributors, and the distributor(s) would learn about requests for the content.

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

## 7.2. User ID transfer

If a request for "https://distributor.example/publisher/page.sxg" comes with the source's or distributor's user ID for the user, either because it's sent with the distributor's cookies or because the source stashes an encoded user ID into either the request's path or a subdomain, the distributor has a few ways to pass that user ID on to the publisher that signed the page:

1. If the distributor has the publisher's signing keys, it can sign a new page with its user ID directly embedded.
2. Otherwise, the publisher can sign lots of copies of their package, and the distributor can choose a particular copy to send a subset of the bits in its user ID to the publisher on each click, which will eventually transfer the whole thing.

To prevent this, the request for a signed exchange needs to omit credentials and block them from appearing in the URL in the same way it would block them from appearing in a cross-origin URL. We're exploring ways the link can mark the request so user agents can take the right counter-measures.

## 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

Author/Change controller: IETF

Specification document(s): Section 3.1 of this document

### 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 3.7 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 5.1.2 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

IANA is requested to register the MIME media type ([IANA.media-types]) for signed exchanges, application/signed-exchange, as follows:

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 5.3.2 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): .sxxg

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

Provisional registration? Yes

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

IANA is requested to register the MIME media type ([IANA.media-types]) for CBOR-format certificate chains, application/cert-chain+cbor, as follows:

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

Provisional registration? Yes

#### 8.8. The cansignhttpexchanges CAA Parameter

There are no IANA considerations for this parameter.

### 9. References

#### 9.1. Normative References

[CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.

[FETCH] WHATWG, "Fetch", July 2020, <<https://fetch.spec.whatwg.org/>>.

[I-D.ietf-httpbis-header-structure] Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", Work in Progress, Internet-Draft, draft-ietf-httpbis-header-structure-19, 3 June 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-header-structure-19.txt>>.

[I-D.ietf-httpbis-variants] Nottingham, M., "HTTP Representation Variants", Work in Progress, Internet-Draft, draft-ietf-httpbis-variants-06, 3 November 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-variants-06.txt>>.

- [I-D.thomson-http-mice]  
Thomson, M. and J. Yasskin, "Merkle Integrity Content Encoding", Work in Progress, Internet-Draft, draft-thomson-http-mice-03, 13 August 2018,  
<<http://www.ietf.org/internet-drafts/draft-thomson-http-mice-03.txt>>.
- [IANA.media-types]  
IANA, "Media Types",  
<<http://www.iana.org/assignments/media-types>>.
- [POSIX] IEEE and The Open Group, "The Open Group Base Specifications Issue 7", value 1003.1-2008, 2016 Edition, name IEEE, 2016,  
<<http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997,  
<<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002,  
<<https://www.rfc-editor.org/info/rfc3230>>.
- [RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004,  
<<https://www.rfc-editor.org/info/rfc3864>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008,  
<<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008,  
<<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6844] Hallam-Baker, P. and R. Stradling, "DNS Certification Authority Authorization (CAA) Resource Record", RFC 6844, DOI 10.17487/RFC6844, January 2013,  
<<https://www.rfc-editor.org/info/rfc6844>>.

- [RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <<https://www.rfc-editor.org/info/rfc6960>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <<https://www.rfc-editor.org/info/rfc6962>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/info/rfc7049>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <<https://www.rfc-editor.org/info/rfc7234>>.
- [RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <<https://www.rfc-editor.org/info/rfc7540>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [URL] WHATWG, "URL", July 2020, <<https://url.spec.whatwg.org/>>.

## 9.2. Informative References

- [BRs] CA/Browser Forum, "Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates", 10 December 2018, <<https://cabforum.org/baseline-requirements-documents/>>.
- [CRLSets] Langley, A., "Revocation checking and Chrome's CRL", 5 February 2012, <<https://www.imperialviolet.org/2012/02/05/crlsets.html>>.
- [I-D.bishop-httpbis-origin-fed-up] Bishop, M. and E. Nygren, "DNS Security with HTTP/2 ORIGIN", Work in Progress, Internet-Draft, draft-bishop-httpbis-origin-fed-up-00, 8 January 2019, <<http://www.ietf.org/internet-drafts/draft-bishop-httpbis-origin-fed-up-00.txt>>.
- [I-D.burke-content-signature] Burke, B., "HTTP Header for digital signatures", Work in Progress, Internet-Draft, draft-burke-content-signature-00, 7 March 2011, <<http://www.ietf.org/internet-drafts/draft-burke-content-signature-00.txt>>.
- [I-D.cavage-http-signatures] Cavage, M. and M. Sporny, "Signing HTTP Messages", Work in Progress, Internet-Draft, draft-cavage-http-signatures-12, 21 October 2019, <<http://www.ietf.org/internet-drafts/draft-cavage-http-signatures-12.txt>>.
- [I-D.ietf-httpbis-cache] Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", Work in Progress, Internet-Draft, draft-ietf-httpbis-cache-10, 12 July 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-cache-10.txt>>.
- [I-D.ietf-httpbis-http2-secondary-certs] Bishop, M., Sullivan, N., and M. Thomson, "Secondary Certificate Authentication in HTTP/2", Work in Progress, Internet-Draft, draft-ietf-httpbis-http2-secondary-certs-06, 14 May 2020, <<http://www.ietf.org/internet-drafts/draft-ietf-httpbis-http2-secondary-certs-06.txt>>.

- [I-D.thomson-http-content-signature]  
Thomson, M., "Content-Signature Header Field for HTTP",  
Work in Progress, Internet-Draft, draft-thomson-http-  
content-signature-00, 2 July 2015, <[http://www.ietf.org/  
internet-drafts/draft-thomson-http-content-signature-  
00.txt](http://www.ietf.org/internet-drafts/draft-thomson-http-content-signature-00.txt)>.
- [I-D.yasskin-httpbis-origin-signed-exchanges-impl]  
Yasskin, J. and K. Ueno, "Signed HTTP Exchanges  
Implementation Checkpoints", Work in Progress, Internet-  
Draft, draft-yasskin-httpbis-origin-signed-exchanges-impl-  
03, 25 July 2019, <[http://www.ietf.org/internet-drafts/  
draft-yasskin-httpbis-origin-signed-exchanges-impl-  
03.txt](http://www.ietf.org/internet-drafts/draft-yasskin-httpbis-origin-signed-exchanges-impl-03.txt)>.
- [I-D.yasskin-wpack-use-cases]  
Yasskin, J., "Use Cases and Requirements for Web  
Packages", Work in Progress, Internet-Draft, draft-  
yasskin-wpack-use-cases-00, 30 October 2019,  
<[http://www.ietf.org/internet-drafts/draft-yasskin-wpack-  
use-cases-00.txt](http://www.ietf.org/internet-drafts/draft-yasskin-wpack-use-cases-00.txt)>.
- [OneCrl] Goodwin, M., "Revoking Intermediate Certificates:  
Introducing OneCRL", 3 March 2015,  
<[https://blog.mozilla.org/security/2015/03/03/revoking-  
intermediate-certificates-introducing-onecrl/](https://blog.mozilla.org/security/2015/03/03/revoking-intermediate-certificates-introducing-onecrl/)>.
- [RFC2965] Kristol, D. and L. Montulli, "HTTP State Management  
Mechanism", RFC 2965, DOI 10.17487/RFC2965, October 2000,  
<<https://www.rfc-editor.org/info/rfc2965>>.
- [RFC5019] Deacon, A. and R. Hurst, "The Lightweight Online  
Certificate Status Protocol (OCSP) Profile for High-Volume  
Environments", RFC 5019, DOI 10.17487/RFC5019, September  
2007, <<https://www.rfc-editor.org/info/rfc5019>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS)  
Extensions: Extension Definitions", RFC 6066,  
DOI 10.17487/RFC6066, January 2011,  
<<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265,  
DOI 10.17487/RFC6265, April 2011,  
<<https://www.rfc-editor.org/info/rfc6265>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454,  
DOI 10.17487/RFC6454, December 2011,  
<<https://www.rfc-editor.org/info/rfc6454>>.

- [RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <<https://www.rfc-editor.org/info/rfc6455>>.
- [RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <<https://www.rfc-editor.org/info/rfc6797>>.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<https://www.rfc-editor.org/info/rfc7235>>.
- [RFC7469] Evans, C., Palmer, C., and R. Slevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <<https://www.rfc-editor.org/info/rfc7469>>.
- [RFC7615] Reschke, J., "HTTP Authentication-Info and Proxy-Authentication-Info Response Header Fields", RFC 7615, DOI 10.17487/RFC7615, September 2015, <<https://www.rfc-editor.org/info/rfc7615>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8053] Oiwa, Y., Watanabe, H., Takagi, H., Maeda, K., Hayashi, T., and Y. Ioku, "HTTP Authentication Extensions for Interactive Clients", RFC 8053, DOI 10.17487/RFC8053, January 2017, <<https://www.rfc-editor.org/info/rfc8053>>.
- [RFC8336] Nottingham, M. and E. Nygren, "The ORIGIN HTTP/2 Frame", RFC 8336, DOI 10.17487/RFC8336, March 2018, <<https://www.rfc-editor.org/info/rfc8336>>.
- [SRI] Akhawe, D., Braun, F., Marier, F., and J. Weinberger, "Subresource Integrity", World Wide Web Consortium Recommendation REC-SRI-20160623, 23 June 2016, <<http://www.w3.org/TR/2016/REC-SRI-20160623>>.
- [W3C.NOTE-OPS-OverHTTP] Hensley, P., Metral, M., Shardanand, U., Converse, D., and M. Myers, "Implementation of OPS Over HTTP", W3C NOTE NOTE-OPS-OverHTTP, 2 June 1997, <<http://www.w3.org/TR/NOTE-OPS-OverHTTP>>.



[W3C.WD-clear-site-data-20171130]

West, M., "Clear Site Data", World Wide Web Consortium WD  
WD-clear-site-data-20171130, 30 November 2017,  
<<https://www.w3.org/TR/2017/WD-clear-site-data-20171130>>.

## 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".

```

```

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 (<https://calendar.perfplanet.com/2013/big-bad-preloader/>) 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 (<https://github.com/mikewest/signature-based-sri>) 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> (<https://github.com/mikewest/signature-based-sri/issues/5>). 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 (<https://www.apple.com/ios/app-store/>) and the Android Play Store (<https://play.google.com/store>) 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> (<https://github.com/WICG/webpackage>) 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 (<https://www.imperialviolet.org/2012/02/05/crlsets.html>), 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 (<https://tlswg.github.io/tls13-spec/draft-ietf-tls-tls13.html#ocsp-and-sct>) 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>
...
```



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-09

- \* No change

draft-08

- \* Improve the privacy considerations.

draft-07

- \* Provisionally register application/signed-exchange and application/cert-chain+cbor.

draft-06

- \* Add a security consideration for future-dated OCSP responses and for stolen private keys.
- \* Define a CAA parameter to opt into certificate issuance.
- \* Limit certificate lifetimes to 90 days.
- \* UTF-8 decode the fallback URL.

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 Andrew Ayer, Devin Mullins, Ilari Liusvaara, John Wilander, Justin Schuh, Mark Nottingham, Mike Bishop, Ryan Sleevi, and Yoav Weiss for comments that improved this draft.

#### Author's Address

Jeffrey Yasskin  
Google

Email: jyasskin@chromium.org