HTTP
Internet-Draft
Obsoletes: 3205 (if approved)
Intended status: Best Current Practice
Expires: May 3, 2020

M. Nottingham
October 31, 2019

Building Protocols with HTTP
draft-ietf-httpbis-bcp56bis-09

Abstract

HTTP is often used as a substrate for other application protocols (a.k.a. HTTP-based APIs). This document specifies best practices for writing specifications that use HTTP to define new application protocols, especially when they are defined for diverse implementation and broad deployment (e.g., in standards efforts).

Note to Readers

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

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/bcp56bis [3].

Status of This Memo

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

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

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

This Internet-Draft will expire on May 3, 2020.

Table of Contents

1. Introduction

HTTP [I-D.ietf-httpbis-semantics] is often used as a substrate for applications other than Web browsing; this is sometimes referred to as creating "HTTP-based APIs", "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; they are 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 is largely because implementations (both client and server) will implement and evolve at different paces. As a result, such an HTTP-based API will need to more carefully consider how extensibility of the service will be handled and how different deployment requirements will be accommodated.

More generally, application protocols using HTTP face a number of design decisions, including:

- Should it define a new 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 the 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.

1.1. Notational Conventions

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

2. Is HTTP Being Used?

Different applications have different goals when using HTTP. The requirements in this document apply when 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", "h2c"), or
- updates or modifies 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 (including but not limited to [I-D.ietf-httpbis-semantics], [I-D.ietf-httpbis-cache], [I-D.ietf-httpbis-messaging], and [RFC7540]).

Note that this document is intended to apply to applications, not generic extensions to HTTP, which follow the requirements in the relevant specification. Furthermore, it is intended for applications defined by IETF specifications, although other standards organisations are encouraged to adhere to its requirements.

2.1. Non-HTTP Protocols

A specification might not use HTTP according to the criteria above and still define an application that relies upon HTTP in some manner. 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 a different set of methods.

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

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 facets of the protocol 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 the payload; often in the body, but also in header fields.

This generic/application-specific split allows a HTTP message to be handled by 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 special-casing 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.

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

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

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

For example, navigating with a link allows a request to be routed to a different server without the overhead of a redirection, thereby supporting deployment across machines well.

It also becomes possible to "mix and match" different applications on the same server, and offers a natural mechanism for extensibility, versioning and capability management, since the document containing the links can also contain information about their targets.

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

3.3. Rich Functionality

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

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

- The ability to interact with the application easily using a Web browser

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

4. Best Practices for 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

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

Because it is a hop-by-hop protocol, a HTTP connection can be handled by implementations that are not controlled by the application; for example, proxies, CDNs, firewalls and so on. Requiring a particular version of HTTP makes it difficult to use in these situations, and harms interoperability for little reason (since HTTP's semantics are stable between protocol versions). Therefore, it is RECOMMENDED that applications using HTTP not 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 full headers, preferably in HTTP/1.1 format. For example:

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


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

[payload here]
```

4.2. Specifying Server Behaviour

The most effective way to specify an application's server-side HTTP behaviours is in terms of the following protocol elements:

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

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

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

For example, an application might specify:

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

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

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

Applications can also specify the use of URI Templates [RFC6570] to allow clients to generate URLs based upon runtime data.

4.3. Specifying Client Behaviour

In general, applications using HTTP ought to align their expectations for client behaviour as closely as possible with that 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 [I-D.ietf-httpbis-rfc6265bis] if they are required.

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

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

application. Applications can encourage the implementation of such features, though.

4.4. Specifying URLs

In HTTP, the server resources that clients interact with are identified with URLs [RFC3986]. As [RFC7320] 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 its URLs; 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 discovery 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, in DNS or mDNS, 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.

Applications MUST NOT define a fixed prefix for its URL paths; for reasons explained in [RFC7320], this is bad practice.

Instead, a specification for such an application can use one of the following strategies:

- Register a Well-Known URI [I-D.nottingham-rfc5785bis] 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 URL Template [RFC6570] or similar mechanism for generating a URL for an entry point. For example, this might be done in a DNS RR, 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), it can be difficult to assure that the new scheme is used consistently.

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

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

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

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

See [RFC7595] for more information about minting new 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 scheme (see Section 4.4.2).

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

See [RFC7605] for further guidance.

4.5. 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 [I-D.ietf-httpbis-semantics]), and are also required to be generic. That means that they need to be potentially applicable to all resources, not just those of one application.

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

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

4.5.1. GET

GET is 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.

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

In some cases, however, GET might be unwieldy for expressing queries, because of the limited syntax of the 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 [I-D.ietf-httpbis-semantics].

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

Applications should not change their state or have other side effects that might be significant to the client, since implementations can and do retry HTTP GET requests that fail. Note that this does not

include logging and similar functions; see [I-D.ietf-httpbis-semantics], Section 7.2.1.

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

4.5.2. OPTIONS

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

However, OPTIONS does have significant limitations:

- It isn't possible to link to the metadata with a simple URL, because OPTIONS is not the default GET method.

- OPTIONS responses are not cacheable, because HTTP caches operate on representations of the resource (i.e., GET and HEAD). If OPTIONS responses are cached separately, their interaction with HTTP cache expiry, secondary keys and other mechanisms needs to be considered.

- OPTIONS is "chatty" - always separating metadata out into a separate request increases the number of requests needed to interact with the application.

- Implementation support for OPTIONS is not universal; some servers do not expose the ability to respond to OPTIONS requests without significant effort.

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

- For server-wide metadata, create a well-known URI [I-D.nottingham-rfc5785bis], or using an already existing one if it's appropriate (e.g., HostMeta [RFC6415]).

- For metadata about a specific resource, create a separate resource and link to it using a Link response header or a link serialised into the representation's body. See [RFC8288]. Note that the Link header is available on HEAD responses, which is useful if the client wants to discover a resource's capabilities before they interact with it.

4.6. 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 intermediaries, as well as server and client implementations. This can happen, for example, when network errors are encountered, a captive portal is present, when an implementation is overloaded, or it thinks it is under attack. As a result, 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 body and/or header fields. [RFC7807] 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 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 [RFC7540] message format.

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

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

4.6.1. Redirection

The 3xx series of status codes specified in Section 9.4 [I-D.ietf-httpbis-semantics] 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 send the request to.

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

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

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

This table summarises their relationships:

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

As noted in [I-D.ietf-httpbis-semantics], a user agent is allowed to automatically follow a 3xx redirect that has a Location response header field, even if they don't understand the semantics of the specific status code. However, they aren't required to do so; therefore, if an application using HTTP desires redirects to be automatically followed, it needs to explicitly specify the circumstances when this is required.

Applications using HTTP 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 ([I-D.ietf-httpbis-semantics]) and Cookie ([I-D.ietf-httpbis-rfc6265bis]) headers will change if the origin (and sometimes path) of the request changes. An application using HTTP should specify if any request headers 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:

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

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

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

See [I-D.ietf-httpbis-semantics], Section 4.1.3 for guidelines to consider when minting new header fields. [I-D.ietf-httpbis-header-structure] provides a common structure for

new header fields, and avoids many issues in their parsing and handling; it is RECOMMENDED that new header fields use it.

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

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

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

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

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

4.8. Defining Message Payloads

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

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

4.9. Leveraging HTTP Caching

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

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 (Section 4.2 of [I-D.ietf-httpbis-cache]) - 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 (Section 5.2.2.8 of [I-D.ietf-httpbis-cache]). The Expires header (Section 5.3 of [I-D.ietf-httpbis-cache]) 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.

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

If caching of a response is not desired, the appropriate response directive is "Cache-Control: no-store". This only need be sent in situations where the response might be cached; see [I-D.ietf-httpbis-cache], Section 3. Note that "Cache-Control: no-cache" allows a response to be stored, just not reused by a cache 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 [I-D.ietf-httpbis-cache], Section 4.2.4, and also [RFC5861] for additional controls over stale content.

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

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 body or in a separate header field. When this happens, the relationship between HTTP caching and that lifetime need to be carefully considered, since the response will be used as long as it is considered fresh.

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 Section 4.2.3 of [I-D.ietf-httpbis-cache]).

One way to address this is to explicitly specify that all responses 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 headers or body, authors should point out that this has implications for caching; in general, such resources need to either make their responses uncacheable (e.g., with the "no-store" cache-control directive defined in [I-D.ietf-httpbis-cache], Section 5.2.2.3) or send the Vary response header ([I-D.ietf-httpbis-semantics], Section 10.1.4) on all responses from that resource (including the "default" response).

For example, this response:

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

[content]
```

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

4.10. Handling Application State

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

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

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

4.11. Client Authentication

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

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

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

4.12. Co-Existing with Web Browsing

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

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

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

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

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

A complete enumeration of such practices is out of scope for this document, but some considerations include:

- Using an application-specific media type in the Content-Type header, and requiring clients to fail if it is not used.

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

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

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

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

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

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

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

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

[content]
```

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

4.13. Maintaining Application Boundaries

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

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

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

To enable this, when specifying the use of Cookies, HTTP authentication realms [I-D.ietf-httpbis-semantics], or other origin-wide HTTP mechanisms, applications using HTTP should not mandate the use of a particular name, but instead let deployments configure them.

Consideration should be given to scoping them to part of the origin, using their specified mechanisms for doing so.

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

4.14. Using Server Push

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

- Server push is hop-by-hop; that is, it is not automatically forwarded by intermediaries. As a result, it might not work easily (or at all) with proxies, reverse proxies, and Content Delivery Networks.

- Server push can have negative performance impact on HTTP when used incorrectly; in particular, if there is contention with resources that have actually been requested by the client.

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

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

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

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

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

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

4.15. Allowing Versioning and Evolution

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

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

- Using a distinct link relation type [RFC8288] to identify a URL for a resource that implements the new functionality.
- Using a distinct media type [RFC6838] to identify formats that enable the new functionality.
- Using a distinct HTTP header field to implement new functionality outside the message body.

5. IANA Considerations

This document has no requirements for IANA.

6. Security Considerations

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

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

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

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

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

Applications that use HTTP in a manner that involves modification of implementations - for example, requiring support for a new 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 mobile code.

Session information includes things like the IP address of the client, TLS session tickets, Cookies, ETags stored in the client's cache, and other stateful mechanisms. Applications are advised to avoid using session mechanisms unless they are unavoidable or necessary for operation, in which case these risks needs to be documented. When they are used, implementations should be encouraged to allow clearing such state.

Fingerprinting uses unique aspects of a client's messages and behaviours to connect disparate requests and connections. For example, the User-Agent request header conveys specific information about the implementation; the Accept-Language request header conveys the users' preferred language. In combination, a number of these markers can be used to uniquely identify a client, impacting its control over its data. As a result, applications are advised to specify that clients should only emit the information they need to function in requests.

Finally, if an application exposes the ability to run mobile code, great care needs to be taken, since any ability to observe its environment can be used as an opportunity to both fingerprint the client and to obtain and manipulate private data (including session information). For example, access to high-resolution timers (even indirectly) can be used to profile the underlying hardware, creating a unique identifier for the system. Applications are advised 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

[I-D.ietf-httpbis-cache]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP Caching", draft-ietf-httpbis-cache-05 (work in progress), July 2019.

[I-D.ietf-httpbis-messaging]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP/1.1 Messaging", draft-ietf-httpbis-messaging-05 (work in progress), July 2019.

[I-D.ietf-httpbis-semantics]
Fielding, R., Nottingham, M., and J. Reschke, "HTTP Semantics", draft-ietf-httpbis-semantics-05 (work in progress), July 2019.

[I-D.nottingham-rfc5785bis]
Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", draft-nottingham-rfc5785bis-11 (work in progress), April 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>.

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

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/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/info/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/info/rfc6838>.

[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>.

[RFC7320] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 7320, DOI 10.17487/RFC7320, July 2014, <https://www.rfc-editor.org/info/rfc7320>.

[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>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

7.2. Informative References

[CSP] West, M., "Content Security Policy Level 3", World Wide Web Consortium WD WD-CSP3-20160913, 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>.

[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.

[I-D.ietf-httpbis-rfc6265bis] Barth, A. and M. West, "Cookies: HTTP State Management Mechanism", draft-ietf-httpbis-rfc6265bis-03 (work in progress), April 2019.

[REFERRER-POLICY]
Eisinger, J. and E. Stark, "Referrer Policy", World Wide Web Consortium CR CR-referrer-policy-20170126, 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/info/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/info/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/info/rfc4918>.

[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>.

[RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale Content", RFC 5861, DOI 10.17487/RFC5861, May 2010, <https://www.rfc-editor.org/info/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/info/rfc6415>.

[RFC6570] 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/info/rfc6570>.

[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>.

[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>.

[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>.

[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/info/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/info/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/info/rfc7616>.

[RFC7617] Reschke, J., "The 'Basic' HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <https://www.rfc-editor.org/info/rfc7617>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

[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>.

[RFC8297] Oku, K., "An HTTP Status Code for Indicating Hints", RFC 8297, DOI 10.17487/RFC8297, December 2017, <https://www.rfc-editor.org/info/rfc8297>.

[SECCTXT] West, M., "Secure Contexts", World Wide Web Consortium CR CR-secure-contexts-20160915, September 2016, <https://www.w3.org/TR/2016/CR-secure-contexts-20160915>.

[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, November 2008, <http://www.w3.org/TR/2008/REC-xml-20081126>.

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/bcp56bis

Appendix A. Changes from RFC 3205

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

Author's Address

Mark Nottingham

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