ï»¿

Internet Engineering Task Force (IETF)
Request for Comments: 9205
BCP: 56
Obsoletes: 3205
Category: Best Current Practice
ISSN: 2070-1721

M. Nottingham

June 2022

Building Protocols with HTTP

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 RFC 3205.

Status of This Memo

This memo documents an Internet Best Current Practice.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on BCPs is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9205.

Table of Contents

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 favour 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 coexist. 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?

Section 2 defines when this document applies, Section 3 surveys the properties of HTTP that are important to preserve, and Section 4 contains best practices for the specification of applications that use HTTP.

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

This document obsoletes [RFC3205] to reflect the 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 ([HTTP] in particular 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 as defined above. For example, an application might wish to avoid re-specifying parts of the message format but might change other aspects of the protocol's operation, or it might want to use application-specific methods.

Doing so permits more freedom to modify protocol operations, but at least a portion of the benefits outlined in Section 3 are lost as most HTTP implementations won't be easily adaptable to these changes. The benefit of mindshare will also 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 and are not specific to a particular context. Application-specific semantics are best expressed in message content and header fields, not status codes or methods (although status codes and methods do have generic semantics that relate to application state).

This split between generic and application-specific semantics allows an HTTP message to be handled by common software (e.g., HTTP servers, intermediaries, client implementations, and caches) without requiring those implementations to understand the application in use. It also allows people to leverage their knowledge of HTTP semantics without needing specialised knowledge of a particular application.

Therefore, applications that use HTTP MUST NOT redefine, 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 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 namespace (or a portion thereof) is exclusively for the use of a single application. This effectively overlays special, application-specific semantics onto that space and precludes other applications from using it.

As explained in [BCP190], 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.

By using links, it also becomes possible to "mix and match" different applications on the same server. The use of links also offers a natural mechanism for extensibility, versioning, and capability management because 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 the affected links so that a fresh copy is always fetched.

See Section 4.4 for more details.

3.3. Rich Functionality

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

* Message framing
* Multiplexing (in HTTP/2 [HTTP/2] and HTTP/3 [HTTP/3])
* Integration with TLS
* Support for intermediaries (proxies, gateways, content delivery networks (CDNs))
* 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

An application that uses HTTP is encouraged to utilise the various features that the protocol offers so that its users receive the maximum benefit from those features and so that the application can be deployed in a variety of situations. This document does not require specific features to be used since the appropriate design trade-offs 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 benefits 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 [HTTP/1.1]. 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, 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 resource 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 require that clients statically support HTTP features that are usually negotiated. 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 [BCP190] 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 which URL scheme(s) the application should be used with and whether to use a dedicated port or to 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 ensures 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 [BCP190]. 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 preclude 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 to mitigate pervasive monitoring attacks [RFC7258].

However, application-specific schemes can also be defined. When defining a URI scheme for an application using HTTP, there are a number of trade-offs 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 commonly occur in HTTP artefacts and are often generated automatically (e.g., in the Location response header field), it can be difficult to ensure 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 ensure 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], HTTP Strict Transport Security (HSTS) [RFC6797], and Cross-Origin Resource Sharing (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. Choosing 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 application-specific 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 <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 and side-effect free linking and underlie 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 the 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 the URI syntax.

While this is not an issue for short queries, it can become one for larger query terms or those that 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 require support for POST queries ought to consider allowing both methods.

Processing of GET requests should not change the application's state or have other side effects that might be significant to the client since implementations can and do retry HTTP GET requests that fail. Furthermore, 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; per [HTTP], Section 9.3.1, content in 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 Web Distributed Authoring and Versioning (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 interactions with the HTTP cache expiry, secondary keys, and other mechanisms need to be considered.

* OPTIONS is "chatty" -- requesting metadata separately 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., host-meta [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; when a captive portal, proxy, or content delivery network is present; or when a server is overloaded or 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 (Bad Request) 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 [HTTP/2] 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 <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] directs 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 change of the request method from POST to GET	301	302
Does not allow change of 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. 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, 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 hyphen.

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 redefined 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 choose (per Section 4.1 of [HTTP-CACHING]) 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 Concise Binary Object Representation (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 the Cache-Control header field, 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 the 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 cache response directive is no-store. Other directives are not necessary, and no-store only needs to be sent in situations where the response might be cached; see [HTTP-CACHING], Section 3. Note that the no-cache directive 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 send the must-revalidate cache directive. 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 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 ensure the intent of the client.

4.11. Making Multiple Requests

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

In HTTP/1 [HTTP/1.1], 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 [HTTP/2] and HTTP/3 [HTTP/3] 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 their processing order. This is because they might be sent over a multiplexed protocol by an intermediary or 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 ensure 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. 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 the private cache directive 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 ensure the intent of the client.

4.13. Coexisting 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 ensure 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 XSS 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 ensure 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 network 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 an 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) that use the same origin server.

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 server. 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 [BCP190] for details).

Therefore, applications using HTTP should strive to allow multiple applications on an origin. Specifically, 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 [HTTP/2], 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 a negative performance impact on HTTP when used incorrectly, particularly 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 [HTTP/2], 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 IANA actions.

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 to 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 websites (and other applications).

Section 4.15 highlights risks of using HTTP/2 server push in a manner other than that 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

[BCP190] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 8820, DOI 10.17487/RFC8820, June 2020, <https://www.rfc-editor.org/rfc/rfc8820>.

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[HTTP-CACHING] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

[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>.

[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>.

[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>.

[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/info/rfc8941>.

[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/info/rfc3986>.

[WEB-LINKING] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/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/info/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/info/rfc6265>.

[CSP] West, M., "Content Security Policy Level 3", W3C Working Draft, June 2021, <https://www.w3.org/TR/2021/WD-CSP3-20210629>.

[FETCH] WHATWG, "Fetch - Living Standard", <https://fetch.spec.whatwg.org>.

[HTML] WHATWG, "HTML - Living Standard", <https://html.spec.whatwg.org>.

[HTTP-PRIORITY] å¥¥ ä¸\200ç©\202 (Oku, K.) and L. Pardue, "Extensible Prioritization Scheme for HTTP", RFC 9218, DOI 10.17487/RFC9218, June 2022, <https://www.rfc-editor.org/info/rfc9218>.

[HTTP/1.1] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/info/rfc9112>.

[HTTP/2] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/info/rfc9113>.

[HTTP/3] Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/info/rfc9114>.

[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/info/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/info/rfc7807>.

[REFERRER-POLICY]
Eisinger, J. and E. Stark, "Referrer Policy", W3C Candidate Recommendation 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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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>.

[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/info/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/info/rfc8949>.

[SECCTXT] West, M., "Secure Contexts", W3C Candidate Recommendation, September 2021, <https://www.w3.org/TR/2021/CRD-secure-contexts-20210918>.

[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/info/rfc6570>.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, M., Maler, E., and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", W3C Recommendation REC-xml-20081126, 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 2000s based on the concerns facing protocol designers at the time. Use of HTTP has changed considerably since then; as a result, this document is substantially different. Consequently, the changes are too numerous to list individually.

Author's Address

Mark Nottingham
Prahran
Australia
Email: mnot@mnot.net
URI: https://www.mnot.net/