HTTP M. Nottingham
Internet-Draft February 12, 2018
Obsoletes: 3205 (if approved)
Intended status: Best Current Practice
Expires: August 16, 2018

On the use of HTTP as a Substrate
draft-ietf-httpbis-bcp56bis-01

Abstract

HTTP is often used as a substrate for other application protocols. This document specifies best practices for these protocols' use of HTTP.

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].

Table of Contents

1. Introduction

HTTP [RFC7230] is often used as a substrate for other application protocols. 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,

- ubiquity 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.

The Internet community has a long tradition of protocol reuse, dating back to the use of Telnet [RFC0854] as a substrate for FTP [RFC0959] and SMTP [RFC2821]. However, layering new protocols over HTTP brings its own set of issues:

- Should an application using HTTP define a new URL scheme? Use new ports?

- Should it use standard HTTP methods and status codes, or define new ones?

- How can the maximum value be extracted from the use of HTTP?

- How does it coexist with other uses of HTTP - especially Web browsing?

- How can interoperability problems and "protocol dead ends" be avoided?

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

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

1.1. Notational Conventions

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

2. Is HTTP Being Used?

Different applications have different goals when using HTTP. In this document, we say an application is _using HTTP_ when any of the following conditions are true:

o The transport port in use is 80 or 443,

o The URL scheme "http" or "https" is used,

o The ALPN protocol ID [RFC7301] generically identifies HTTP (e.g., "http/1.1", "h2", "h2c"), or

o The message formats described in [RFC7230] and/or [RFC7540] are used in conjunction with the IANA registries defined for HTTP.

When an application is using HTTP, all of the requirements of the HTTP protocol suite (including but not limited to [RFC7230], [RFC7231], [RFC7232], [RFC7233], [RFC7234], [RFC7235] and [RFC7540]) are in force.

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

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

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

3. What's Important About HTTP

There are many ways that applications using HTTP are defined and deployed, and sometimes they are brought to the IETF for standardisation. In that process, what might be workable for deployment in a limited fashion isn't appropriate for standardisation and the corresponding broader deployment.

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

3.1. Generic Semantics

When writing an application's 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 `200 OK` response means that the widget has successfully been updated.

This sort of specification is bad practice, because it is adding new semantics to HTTP's status codes and methods, respectively; a recipient - whether it's an origin server, client library, intermediary or cache - now has to know these extra semantics to understand the message.

Some applications even require specific behaviours, such as:

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

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

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

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

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

See Section 4.2 for details.

3.2. Links

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

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

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

Using runtime links in this fashion has a number of other benefits. 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 becomes possible to "mix" different applications on the same server, and offers a natural path for extensibility, versioning and capability management.

3.3. Getting Value from HTTP

The simplest possible use of HTTP is to POST data to a single URL, thereby effectively tunnelling through the protocol.

This "RPC" style of communication does get some benefit from using HTTP - namely, message framing and the availability of implementations - but fails to realise many others:

- o Caching for server scalability, latency and bandwidth reduction, and reliability;
- o Authentication and access control;
- o Automatic redirection;
- o Partial content to selectively request part of a response;

- Natural support for extensions and versioning through protocol extension; and

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

Using such a high-level protocol to tunnel simple semantics has downsides too; because of its more advanced capabilities, breadth of deployment and age, HTTP's complexity can cause interoperability problems that could be avoided by using a simpler substrate (e.g., WebSockets [RFC6455], if browser support is necessary, or TCP [RFC0793] if not), or making the application be _based upon HTTP_, instead of using it (as defined in Section 2).

Applications that use HTTP are encouraged to accommodate the various features that the protocol offers, so that their users receive the maximum benefit from it. 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 will help make them available.

4. Best Practices for Using HTTP

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

4.1. Specifying the Use of HTTP

When specifying the use of HTTP, an application SHOULD use [RFC7230] as the primary reference; it is not necessary to reference all of the specifications in the HTTP suite unless there are specific reasons to do so (e.g., a particular feature is called out).

Applications using HTTP MAY specify a minimum version to be supported (HTTP/1.1 is suggested), and MUST NOT specify a maximum version.

Likewise, applications need not specify what HTTP mechanisms - such as redirection, caching, authentication, proxy authentication, and so on - are to be supported. Full featured support for HTTP SHOULD be taken for granted in servers and clients, and the application's function SHOULD degrade gracefully if they are not (although this might be achieved by informing the user that their task cannot be completed).

For example, an application can specify that it uses HTTP like this:

Foo Application uses HTTP {{RFC7230}}. Implementations MUST support HTTP/1.1, and MAY support later versions. Support for common HTTP mechanisms such as redirection and caching are assumed.

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

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

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

[payload here]
```

4.2. Defining HTTP Resources

HTTP Applications SHOULD focus on defining the following application-specific protocol elements:

- o Media types [RFC6838], often based upon a format convention such as JSON [RFC7159],
- o HTTP header fields, as per Section 4.6, and
- o The behaviour of resources, as identified by link relations [RFC5988].

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

- o Retrieval of their state using GET, in one or more formats identified by media type;
- o Resource creation or update using POST or PUT, with an appropriately identified request body format;
- o Data processing using POST and identified request and response body format(s); and
- o 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 {{RFC7159}} format representing the state of a Widget. It contains links to related information in the link indicated by the Link header field value with the "example-other-info" link relation type.

4.3. HTTP URLs

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

In other words, applications that use HTTP MUST NOT associate application semantics with specific URL paths on arbitrary servers. Doing so inappropriately conflates the identity of the resource (its URL) with the capabilities that resource supports, bringing about many of the same interoperability problems that [RFC4367] warns of.

For example, specifying that a "GET to the URL /foo retrieves a bar document" is bad practice. Likewise, specifying "The widget API is at the path /bar" violates [RFC7320].

Instead, applications that use HTTP are encouraged to ensure that URLs are discovered at runtime, allowing HTTP-based services to describe their own capabilities. One way to do this is to use typed links [RFC5988] to convey the URIs that are in use, as well as the semantics of the resources that they identify. See Section 4.2 for details.

4.3.1. Initial URL Discovery

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

Applications that use HTTP SHOULD allow an arbitrary URL to be used as that entry point. For example, rather than specifying "the

initial document is at "/foo/v1", they should allow a deployment to use any URL as the entry point for the application.

In cases where doing so is impractical (e.g., it is not possible to convey a whole URL, but only a hostname) standard applications that use HTTP can request a well-known URL [RFC5785] as an entry point.

4.3.2. URL Schemes

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

However, application-specific schemes can be defined as well.

When defining an URL scheme for an application using HTTP, there are a number of tradeoffs and caveats to keep in mind:

- Unmodified Web browsers will not support the new scheme. While it is possible to register new URL schemes with Web browsers (e.g. registerProtocolHandler() in [HTML5] Section 8.7.1.3, 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 and are generated in HTTP artefacts commonly, often without human intervention (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 [RFC6265], authentication [RFC7235], caching [RFC7234], 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 [W3C.CR-secure-contexts-20160915] will likely treat a new scheme as insecure.

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

4.3.3. Transport Ports

Applications that use HTTP 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).

In either case, non-default ports will need 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 4.3.2).

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.

4.4. 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 with IETF Review (see [RFC7232]), 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]) has defined non-generic methods, [RFC7231] now forbids this.

When it is believed that a new method is required, authors 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. HTTP Status Codes

Applications that use HTTP MUST only use registered HTTP status codes.

As with methods, new HTTP status codes are rare, and required (by [RFC7231]) to be registered with IETF review. Similarly, HTTP status codes are generic; they are required (by [RFC7231]) to be potentially applicable to all resources, not just to those of one application.

When it is believed that a new status code is required, authors 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.

Status codes' primary function is to convey HTTP semantics for the benefit of generic HTTP software, not application-specific semantics. Therefore, applications MUST NOT specify additional semantics or refine existing semantics for status codes.

In particular, specifying that a particular status code has a specific meaning in the context of an application is harmful, as these are not generic semantics, since the consumer needs to be in the context of the application to understand them.

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

Typically, applications using HTTP will convey application-specific information in the message body and/or HTTP header fields, not the status code.

Specifications sometimes also create a "laundry list" of potential status codes, in an effort to be helpful. The problem with doing so is that such a list is never complete; for example, if a network proxy is interposed, the client might encounter a "407 Proxy Authentication Required" response; or, if the server is rate limiting the client, it might receive a "429 Too Many Requests" response.

Since the list of HTTP status codes can be added to, it's safer to refer to it directly, and point out that clients SHOULD be able to handle all applicable protocol elements 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).

4.6. HTTP Header Fields

Applications that use HTTP MAY define new HTTP header fields, following the advice in [RFC7231], Section 8.3.1.

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).

If none of these motivations apply, using a header field is NOT RECOMMENDED.

New header fields MUST be registered, as per [RFC7231] and [RFC3864].

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

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

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

See Section 4.8 for requirements regarding header fields that carry application state (e.g,. Cookie).

4.7. Defining Message Payloads

There are many potential formats for payloads; for example, JSON [RFC8259] and XML [W3C.REC-xml-20081126]. 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.

4.8. Authentication and Application State

Applications that use HTTP MAY use stateful cookies [RFC6265] to identify a client and/or store client-specific data to contextualise requests.

If it is only necessary to identify clients, applications that use HTTP MAY use HTTP authentication [RFC7235]; if either of the Basic [RFC7617] or Digest [RFC7616] authentication schemes is used, it MUST NOT be used with the 'http' URL scheme.

In either case, it is important to carefully specify the scoping and use of these mechanisms; if they expose 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.

4.9. Co-Existing with Web Browsing

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

This means that all such applications 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 making that request by a HTML form on an arbitrary Web site.

Or, if a resource reflects data from the request into a response, that can be used to perform a Cross-Site Scripting attack on Web browsers directed to it.

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 _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. External resources are numerous; e.g., https://www.owasp.org/index.php/OWASP_Guide_Project [4].

4.10. Co-Existing with Other Applications

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 [RFC6265] 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 collisions in cookie name.

As a result, when specifying the use of Cookies, HTTP authentication [RFC7235], or other origin-wide HTTP mechanisms, applications using HTTP SHOULD NOT mandate the use of a particular identifier, but instead let deployments configure them.

Note that dedicating a hostname to a single application is not a solution to the issues above; see [RFC7320].

Modern Web browsers constrain the ability of content from one origin to access resources from another, to avoid the "confused deputy" problem. As a result, applications that wish to expose cross-origin data to browsers will need to implement [W3C.REC-cors-20140116].

5. IANA Considerations

This document has no requirements for IANA.

6. Security Considerations

Section 4.8 discusses the impact of using stateful mechanisms in the protocol as ambient authority, and suggests a mitigation.

Section 4.3.2 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 4.9 highlights the implications of Web browsers' capabilities on applications that use HTTP.

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

7. References

7.1. Normative References

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

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <https://www.rfc-editor.org/info/rfc5988>.

[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>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <https://www.rfc-editor.org/info/rfc7232>.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <https://www.rfc-editor.org/info/rfc7233>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <https://www.rfc-editor.org/info/rfc7235>.

[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>.

[W3C.REC-cors-20140116]
Kesteren, A., "Cross-Origin Resource Sharing", World Wide Web Consortium Recommendation REC-cors-20140116, January 2014, <http://www.w3.org/TR/2014/REC-cors-20140116>.

7.2. Informative References

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

[HTML5] WHATWG, "HTML - Living Standard", n.d., <https://html.spec.whatwg.org>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <https://www.rfc-editor.org/info/rfc854>.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, DOI 10.17487/RFC0959, October 1985, <https://www.rfc-editor.org/info/rfc959>.

[RFC2821] Klensin, J., Ed., "Simple Mail Transfer Protocol", RFC 2821, DOI 10.17487/RFC2821, April 2001, <https://www.rfc-editor.org/info/rfc2821>.

[RFC4367] Rosenberg, J., Ed. and IAB, "What's in a Name: False Assumptions about DNS Names", RFC 4367, DOI 10.17487/RFC4367, February 2006, <https://www.rfc-editor.org/info/rfc4367>.

[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>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <https://www.rfc-editor.org/info/rfc5785>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/info/rfc6455>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

[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>.

[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>.

[W3C.CR-secure-contexts-20160915]
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>.

[W3C.REC-xml-20081126]
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

[4] https://www.owasp.org/index.php/OWASP_Guide_Project

Appendix A. Changes from RFC3205

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/