Structured Headers for HTTP
draft-ietf-httpbis-header-structure-02

Abstract

   This document describes Structured Headers, a way of simplifying HTTP
   header field definition and parsing. It is intended for use by new
   specifications of HTTP header fields. This includes revisions of
   existing specifications when doing so does not cause interoperability
   issues.

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].

   RFC EDITOR: please remove this section before publication

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

Status of This Memo

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

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

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

   This Internet-Draft will expire on May 31, 2018.

1. Introduction

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

   Likewise, bespoke parsers often need to be written for specific HTTP
   headers, because each has slightly different handling of what looks
   like common syntax.

   This document introduces structured HTTP header field values
   (hereafter, Structured Headers) to address these problems.
   Structured Headers define a generic, abstract model for data, along
   with a concrete serialisation for expressing that model in textual
   HTTP headers, as used by HTTP/1 [[RFC7230]()] and HTTP/2 [[RFC7540]()].

   HTTP headers that are defined as Structured Headers use the types
   defined in this specification to define their syntax and basic
   handling rules, thereby simplifying both their definition and
   parsing.

   Additionally, future versions of HTTP can define alternative
   serialisations of the abstract model of Structured Headers, allowing
   headers that use it to be transmitted more efficiently without being
   redefined.

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

   To specify a header field that uses Structured Headers, see
   [Section 2]().

Section 4 defines a number of abstract data types that can be used in Structured Headers, of which only three are allowed at the "top" level: lists, dictionaries, or items.

Those abstract types can be serialised into textual headers - such as those used in HTTP/1 and HTTP/2 - using the algorithms described in Section 3.

1.1. Notational Conventions

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

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234], including the DIGIT, ALPHA and DQUOTE rules from that document. It also includes the OWS rule from [RFC7230].

2. Specifying Structured Headers

HTTP headers that use Structured Headers need to be defined to do so explicitly; recipients and generators need to know that the requirements of this document are in effect. The simplest way to do that is by referencing this document in its definition.

The field's definition will also need to specify the field-value's allowed syntax, in terms of the types described in Section 4, along with their associated semantics.

Field definitions MUST NOT relax or otherwise modify the requirements of this specification; doing so would preclude handling by generic software.

However, field definitions are encouraged to clearly state additional constraints upon the syntax, as well as the consequences when those constraints are violated.

For example:

FooExample Header

The FooExample HTTP header field conveys a list of numbers about how much Foo the sender has.

FooExample is a Structured header [RFCxxxx]. Its value MUST be a dictionary ([RFCxxxx], Section Y.Y).

The dictionary MUST contain:

* A member whose key is "foo", and whose value is an integer ([RFCxxxx], Section Y.Y), indicating the number of foos in the message.
* A member whose key is "bar", and whose value is a string ([RFCxxxx], Section Y.Y), conveying the characteristic bar-ness of the message.

If the parsed header field does not contain both, it MUST be ignored.

Note that empty header field values are not allowed by the syntax, and therefore will be considered errors.

3. Parsing Requirements for Textual Headers

When a receiving implementation parses textual HTTP header fields (e.g., in HTTP/1 or HTTP/2) that are known to be Structured Headers, it is important that care be taken, as there are a number of edge cases that can cause interoperability or even security problems. This section specifies the algorithm for doing so.

Given an ASCII string input_string that represents the chosen header's field-value, return the parsed header value. Note that input_string may incorporate multiple header lines combined into one comma-separated field-value, as per [RFC7230], Section 3.2.2.

1. Discard any OWS from the beginning of input_string.

2. If the field-value is defined to be a dictionary, return the result of Parsing a Dictionary from Textual headers (Section 4.7).

3. If the field-value is defined to be a list, return the result of Parsing a List from Textual Headers (Section 4.8).

4. If the field-value is defined to be a parameterised label, return the result of Parsing a Parameterised Label from Textual headers (Section 4.4).

5. Otherwise, return the result of Parsing an Item from Textual
    Headers (Section 4.6).

Note that in the case of lists and dictionaries, this has the effect
of combining multiple instances of the header field into one.
However, for singular items and parameterised labels, it has the
effect of selecting the first value and ignoring any subsequent
instances of the field, as well as extraneous text afterwards.

Additionally, note that the effect of the parsing algorithms as
specified is generally intolerant of syntax errors; if one is
encountered, the typical response is to throw an error, thereby
discarding the entire header field value. This includes any non-
ASCII characters in input_string.

4. Structured Header Data Types

This section defines the abstract value types that can be composed
into Structured Headers, along with the textual HTTP serialisations
of them.

4.1. Numbers

Abstractly, numbers are integers with an optional fractional part.
They have a maximum of fifteen digits available to be used in one or
both of the parts, as reflected in the ABNF below; this allows them
to be stored as IEEE 754 double precision numbers (binary64)
([IEEE754]).

The textual HTTP serialisation of numbers allows a maximum of fifteen
digits between the integer and fractional part, along with an
optional "-" indicating negative numbers.

```
   number   = ["-"] ( "." 1*15DIGIT /
                 DIGIT "." 1*14DIGIT /
                2DIGIT "." 1*13DIGIT /
                3DIGIT "." 1*12DIGIT /
                4DIGIT "." 1*11DIGIT /
                5DIGIT "." 1*10DIGIT /
                6DIGIT "." 1*9DIGIT /
                7DIGIT "." 1*8DIGIT /
                8DIGIT "." 1*7DIGIT /
                9DIGIT "." 1*6DIGIT /
               10DIGIT "." 1*5DIGIT /
               11DIGIT "." 1*4DIGIT /
               12DIGIT "." 1*3DIGIT /
               13DIGIT "." 1*2DIGIT /
               14DIGIT "." 1DIGIT /
               15DIGIT )

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

   integer and unsigned are defined as conveniences to specification
   authors; if their use is specified and their ABNF is not matched, a
   parser MUST consider it to be invalid.

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

   ExampleNumberHeader: 4.5

4.1.1. Parsing Numbers from Textual Headers

   TBD

4.2. Strings

   Abstractly, strings are ASCII strings [RFC0020], excluding control
   characters (i.e., the range 0x20 to 0x7E). Note that this excludes
   tabs, newlines and carriage returns. They may be at most 1024
   characters long.

   The textual HTTP serialisation of strings uses a backslash ("") to
   escape double quotes and backslashes in strings.

```
   string    = DQUOTE 1*1024(char) DQUOTE
   char      = unescaped / escape ( DQUOTE / "\" )
   unescaped = %x20-21 / %x23-5B / %x5D-7E
   escape    = "\"
```

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

ExampleStringHeader: "hello world"

Note that strings only use DQUOTE as a delimiter; single quotes do
not delimit strings. Furthermore, only DQUOTE and "" can be escaped;
other sequences MUST generate an error.

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

When it is necessary for a field value to convey non-ASCII string
content, binary content (Section 4.5) SHOULD be specified, along with
a character encoding (most likely, UTF-8).

4.2.1. Parsing a String from Textual Headers

Given an ASCII string input_string, return an unquoted string.
input_string is modified to remove the parsed value.

1. Let output_string be an empty string.

2. If the first character of input_string is not DQUOTE, throw an
    error.

3. Discard the first character of input_string.

4. If input_string contains more than 1025 characters, throw an
    error.

5. While input_string is not empty:

    1. Let char be the result of removing the first character of
        input_string.

    2. If char is a backslash ("\"):

        1. If input_string is now empty, throw an error.

        2. Else:

            1. Let next_char be the result of removing the first
                character of input_string.

            2. If next_char is not DQUOTE or "\", throw an error.

3. Append next_char to output_string.

3. Else, if char is DQUOTE, remove the first character of
    input_string and return output_string.

4. Else, append char to output_string.

6. Otherwise, throw an error.

4.3. Labels

Labels are short (up to 256 characters) textual identifiers; their
abstract model is identical to their expression in the textual HTTP
serialisation.

```
label = lcalpha *255( lcalpha / DIGIT / "_" / "-"/ "*" / "/" )
lcalpha = %x61-7A ; a-z
```

Note that labels can only contain lowercase letters.

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

ExampleLabelHeader: foo/bar

4.3.1. Parsing a Label from Textual Headers

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

1. If input_string contains more than 256 characters, throw an
    error.

2. If the first character of input_string is not lcalpha, throw an
    error.

3. Let output_string be an empty string.

4. While input_string is not empty:

    1. Let char be the result of removing the first character of
        input_string.

    2. If char is not one of lcalpha, DIGIT, "_", "-", "*" or "/":

        1. Prepend char to input_string.

        2. Return output_string.

3. Append char to output_string.

5. Return output_string.

4.4. Parameterised Labels

Parameterised Labels are labels (Section 4.3) with up to 256
parameters; each parameter has a label and an optional value that is
an item (Section 4.6). Ordering between parameters is not
significant, and duplicate parameters MUST be considered an error.

The textual HTTP serialisation uses semicolons (";") to delimit the
parameters from each other, and equals ("=") to delimit the parameter
name from its value.

parameterised = label *256(OWS ";" OWS label ["=" item])

For example,

ExampleParamHeader: abc; a=1; b=2; c

4.4.1. Parsing a Parameterised Label from Textual Headers

Given an ASCII string input_string, return a label with an mapping of
parameters. input_string is modified to remove the parsed value.

1. Let primary_label be the result of Parsing a Label from Textual
    Headers (Section 4.3) from input_string.

2. Let parameters be an empty mapping.

3. In a loop:

    1.   Consume any OWS from the beginning of input_string.

    2.   If the first character of input_string is not ";", exit the
         loop.

    3.   Consume a ";" character from the beginning of input_string.

    4.   Consume any OWS from the beginning of input_string.

    5.   let param_name be the result of Parsing a Label from Textual
         Headers (Section 4.3) from input_string.

    6.   If param_name is already present in parameters, throw an
         error.

7.   Let param_value be a null value.

8.   If the first character of input_string is "=":

1.   Consume the "=" character at the beginning of
input_string.

2.   Let param_value be the result of Parsing an Item from
Textual Headers ([Section 4.6](#)) from input_string.

9.   If parameters has more than 255 members, throw an error.

10. Add param_name to parameters with the value param_value.

4. Return the tuple (primary_label, parameters).

[4.5](#). Binary Content

Arbitrary binary content up to 16K in size can be conveyed in
Structured Headers.

The textual HTTP serialisation indicates their presence by a leading
"*", with the data encoded using Base 64 Encoding [[RFC4648](#)], without
padding (as "=" might be confused with the use of dictionaries).

```
binary = "*" 1*21846(base64)
base64 = ALPHA / DIGIT / "+" / "/"
```

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

ExampleBinaryHeader: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg

[4.5.1](#). Parsing Binary Content from Textual Headers

Given an ASCII string input_string, return binary content.
input_string is modified to remove the parsed value.

1. If the first character of input_string is not "*", throw an
error.

2. Discard the first character of input_string.

3. Let b64_content be the result of removing content of input_string
up to but not including the first character that is not in ALPHA,
DIGIT, "+" or "/".

4. Let binary_content be the result of Base 64 Decoding [RFC4648]
    b64_content, synthesising padding if necessary. If an error is
    encountered, throw it.

5. Return binary_content.

4.6. Items

An item is can be a number (Section 4.1), string (Section 4.2), label
(Section 4.3) or binary content (Section 4.5).

item = number / string / label / binary

4.6.1. Parsing an Item from Textual Headers

Given an ASCII string input_string, return an item. input_string is
modified to remove the parsed value.

1. Discard any OWS from the beginning of input_string.

2. If the first character of input_string is a "-" or a DIGIT,
    process input_string as a number (Section 4.1) and return the
    result, throwing any errors encountered.

3. If the first character of input_string is a DQUOTE, process
    input_string as a string (Section 4.2) and return the result,
    throwing any errors encountered.

4. If the first character of input_string is "*", process
    input_string as binary content (Section 4.5) and return the
    result, throwing any errors encountered.

5. If the first character of input_string is an lcalpha, process
    input_string as a label (Section 4.3) and return the result,
    throwing any errors encountered.

6. Otherwise, throw an error.

4.7. Dictionaries

Dictionaries are unordered maps of key-value pairs, where the keys
are labels (Section 4.3) and the values are items (Section 4.6).
There can be between 1 and 1024 members, and keys are required to be
unique.

In the textual HTTP serialisation, keys and values are separated by
"=" (without whitespace), and key/value pairs are separated by a
comma with optional whitespace.

```
dictionary = label "=" item *1023( OWS "," OWS label "=" item )
```

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

```
ExampleDictHeader: foo=1.23, en="Applepie", da=*w4ZibGV0w6ZydGUK
```

Typically, a header field specification will define the semantics of individual keys, as well as whether their presence is required or optional. Recipients MUST ignore keys that are undefined or unknown, unless the header field's specification specifically disallows them.

4.7.1. Parsing a Dictionary from Textual Headers

Given an ASCII string input_string, return a mapping of (label, item). input_string is modified to remove the parsed value.

1. Let dictionary be an empty mapping.

2. While input_string is not empty:

    1. Let this_key be the result of running Parse Label from Textual Headers (Section 4.3) with input_string. If an error is encountered, throw it.

    2. If dictionary already contains this_key, raise an error.

    3. Consume a "=" from input_string; if none is present, raise an error.

    4. Let this_value be the result of running Parse Item from Textual Headers (Section 4.6) with input_string. If an error is encountered, throw it.

    5. Add key this_key with value this_value to dictionary.

    6. Discard any leading OWS from input_string.

    7. If input_string is empty, return dictionary.

    8. Consume a COMMA from input_string; if no comma is present, raise an error.

    9. Discard any leading OWS from input_string.

3. Return dictionary.

4.8. Lists

Lists are arrays of items (Section 4.6) or parameterised labels
(Section 4.4, with one to 1024 members.

In the textual HTTP serialisation, each member is separated by a
comma and optional whitespace.

```
list = list_member 1*1024( OWS "," OWS list_member )
list_member = item / parameterised
```

For example, a header field whose value is defined as a list of
labels could look like:

```
ExampleLabelListHeader: foo, bar, baz_45
```

and a header field whose value is defined as a list of parameterised
labels could look like:

```
ExampleParamListHeader: abc/def; g="hi";j, klm/nop
```

4.8.1. Parsing a List from Textual Headers

Given an ASCII string input_string, return a list of items.
input_string is modified to remove the parsed value.

1. Let items be an empty array.

2. While input_string is not empty:

    1. Let item be the result of running Parse Item from Textual
        Headers (Section 4.6) with input_string. If an error is
        encountered, throw it.

    2. Append item to items.

    3. Discard any leading OWS from input_string.

    4. If input_string is empty, return items.

    5. Consume a COMMA from input_string; if no comma is present,
        raise an error.

    6. Discard any leading OWS from input_string.

3. Return items.

5. IANA Considerations

This draft has no actions for IANA.

6. Security Considerations

TBD

7. References

7.1. Normative References

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80,
           RFC 20, DOI 10.17487/RFC0020, October 1969,
           <https://www.rfc-editor.org/info/rfc20>.

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

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data
           Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006,
           <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax
           Specifications: ABNF", STD 68, RFC 5234,
           DOI 10.17487/RFC5234, January 2008,
           <https://www.rfc-editor.org/info/rfc5234>.

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
           2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,
           May 2017, <https://www.rfc-editor.org/info/rfc8174>.

7.2. Informative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", 2008,
           <http://grouper.ieee.org/groups/754/>.

[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>.

   [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>.

7.3. URIs

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

   [2] https://httpwg.github.io/

   [3] https://github.com/httpwg/http-extensions/labels/header-structure

Appendix A. Changes

A.1. Since draft-ietf-httpbis-header-structure-01

   Replaced with draft-nottingham-structured-headers.

A.2. Since draft-ietf-httpbis-header-structure-00

   Added signed 64bit integer type.

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

   Change h1_blob delimiter to ":" since "'" is valid t_char

Authors' Addresses

   Mark Nottingham
   Fastly

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

   Poul-Henning Kamp
   The Varnish Cache Project

   Email: phk@varnish-cache.org