HTTP
Internet-Draft
Intended status: Standards Track
Expires: January 3, 2019

M. Nottingham
Fastly
P-H. Kamp
The Varnish Cache Project

July 2, 2018

Structured Headers for HTTP
draft-ietf-httpbis-header-structure-07

Abstract

This document describes a set of data types and algorithms associated with them that are intended to make it easier and safer to define and handle HTTP header fields. It is intended for use by new specifications of HTTP header fields as well as revisions of existing header field specifications when doing so does not cause interoperability issues.

Note to Readers

RFC EDITOR: please remove this section before publication

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

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

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

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

Status of This Memo

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Table of Contents

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.

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

This document introduces a set of common data structures for use in HTTP header field values to address these problems. In particular, it defines a generic, abstract model for header field values, along with a concrete serialisation for expressing that model in HTTP/1 [RFC7230] header fields.

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 by specification writers and handling by implementations.

Additionally, future versions of HTTP can define alternative serialisations of the abstract model of these structures, 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 is a Structured Header, see Section 2.

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

Those abstract types can be serialised into and parsed from textual headers - such as those used in HTTP/1 - using the algorithms described in Section 4.

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 VCHAR, DIGIT, ALPHA and DQUOTE rules from that document. It also includes the OWS rule from [RFC7230].

This document uses algorithms to specify parsing and serialisation behaviours, and ABNF to illustrate expected syntax.

For parsing, implementations MUST follow the algorithms, but MAY vary in implementation so as the behaviours are indistinguishable from specified behaviour. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence.

For serialisation, the ABNF illustrates the range of acceptable wire representations with as much fidelity as possible, and the algorithms define the recommended way to produce them. Implementations MAY vary from the specified behaviour so long as the output still matches the ABNF.

2. Defining New Structured Headers

To define a HTTP header as a structured header, its specification needs to:

- Reference this specification. Recipients and generators of the header need to know that the requirements of this document are in effect.

- Specify the header field's allowed syntax for values, in terms of the types described in Section 3, along with their associated semantics. Syntax definitions are encouraged to use the ABNF rules beginning with "sh-" defined in this specification.

- Specify any additional constraints upon the syntax of the structured used, as well as the consequences when those constraints are violated. When Structured Headers parsing fails,

the header is discarded (see Section 4.2); in most situations, header-specific constraints should do likewise.

Note that a header field definition cannot relax the requirements of a structure or its processing; they can only add additional constraints, because doing so would preclude handling by generic software.

For example:

```
# Foo-Example Header

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

Foo-Example is a Structured Header [RFCxxxx]. Its value MUST be a
dictionary ([RFCxxxx], Section Y.Y). Its ABNF is:

  Foo-Example = sh-dictionary

The dictionary MUST contain:

* Exactly one member whose key is "foo", and whose value is an
  integer ([RFCxxxx], Section Y.Y), indicating the number of foos
  in the message.
* Exactly one member whose key is "barUrls", and whose value is a
  string ([RFCxxxx], Section Y.Y), conveying the Bar URLs for the
  message. See below for processing requirements.

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

"foo" MUST be between 0 and 10, inclusive; other values MUST cause
the header to be ignored.

"barUrls" contains a space-separated list of URI-references
([RFC3986], Section 4.1):

   barURLs = URI-reference *( 1*SP URI-reference )

If a member of barURLs is not a valid URI-reference, it MUST cause
that value to be ignored.

If a member of barURLs is a relative reference ([RFC3986],
Section 4.2), it MUST be resolved ([RFC3986], Section 5) before
being used.
```

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

3. Structured Header Data Types

This section defines the abstract value types that can be composed into Structured Headers. The ABNF provided represents the on-wire format in HTTP/1.

3.1. Dictionaries

Dictionaries are unordered maps of key-value pairs, where the keys are identifiers (Section 3.8) and the values are items (Section 3.4). There can be one or more members, and keys are required to be unique.

The ABNF for dictionaries is:

```
sh-dictionary  = dict-member *( OWS "," OWS dict-member )
dict-member    = member-name "=" member-value
member-name    = identifier
member-value   = sh-item
```

In HTTP/1, keys and values are separated by "=" (without whitespace), and key/value pairs are separated by a comma with optional whitespace. For example:

```
Example-DictHeader: 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.

Parsers MUST support dictionaries containing at least 1024 key/value pairs.

3.2. Lists

Lists are arrays of items (Section 3.4) with one or more members.

The ABNF for lists is:

```
sh-list     = list-member *( OWS "," OWS list-member )
list-member = sh-item
```

In HTTP/1, each member is separated by a comma and optional whitespace. For example, a header field whose value is defined as a list of strings could look like:

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

Header specifications can constrain the types of individual values if necessary.

Parsers MUST support lists containing at least 1024 members.

3.3. Parameterised Lists

Parameterised Lists are arrays of a parameterised identifiers.

A parameterised identifier is an identifier (Section 3.8) with an optional set of parameters, each parameter having a identifier and an optional value that is an item (Section 3.4). Ordering between parameters is not significant, and duplicate parameters MUST cause parsing to fail.

The ABNF for parameterised lists is:

```
sh-param-list = param-id *( OWS "," OWS param-id )
param-id      = identifier *parameter
parameter     = OWS ";" OWS param-name [ "=" param-value ]
param-name    = identifier
param-value   = sh-item
```

In HTTP/1, each param-id is separated by a comma and optional whitespace (as in Lists), and the parameters are separated by semicolons. For example:

```
Example-ParamListHeader: abc_123;a=1;b=2; cdef_456, ghi;q="9";r=w
```

Parsers MUST support parameterised lists containing at least 1024 members, and support members with at least 256 parameters.

3.4. Items

An item is can be a integer (Section 3.5), float (Section 3.6), string (Section 3.7), or binary content (Section 3.9).

The ABNF for items is:

```
sh-item = sh-integer / sh-float / sh-string / sh-binary
```

3.5. Integers

Integers have a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive (i.e., a 64-bit signed integer).

The ABNF for integers is:

```
sh-integer = ["-"] 1*19DIGIT
```

For example:

```
Example-IntegerHeader: 42
```

3.6. Floats

Floats are integers with a fractional part, that can be stored as IEEE 754 double precision numbers (binary64) ([IEEE754]).

The ABNF for floats is:

```
sh-float    = ["-"] (
             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 )
```

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

```
Example-FloatHeader: 4.5
```

3.7. Strings

Strings are zero or more printable ASCII [RFC0020] characters (i.e., the range 0x20 to 0x7E). Note that this excludes tabs, newlines, carriage returns, etc.

The ABNF for strings is:

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

In HTTP/1 headers, strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

```
Example-StringHeader: "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 cause parsing to fail.

Unicode is not directly supported in this document, 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 3.9) SHOULD be specified, along with a character encoding (preferably, UTF-8).

Parsers MUST support strings with at least 1024 characters.

3.8. Identifiers

Identifiers are short textual identifiers; their abstract model is identical to their expression in the textual HTTP serialisation. Parsers MUST support identifiers with at least 64 characters.

The ABNF for identifiers is:

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

Note that identifiers can only contain lowercase letters.

3.9. Binary Content

Arbitrary binary content can be conveyed in Structured Headers.

The ABNF for binary content is:

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

In HTTP/1 headers, binary content is delimited with asterisks and encoded using base64 ([RFC4648], Section 4). For example:

```
Example-BinaryHdr: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==*
```

Parsers MUST support binary content with at least 16384 octets after decoding.

4. Structured Headers in HTTP/1

This section defines how to serialise and parse Structured Headers in HTTP/1 textual header fields, and protocols compatible with them (e.g., in HTTP/2 [RFC7540] before HPACK [RFC7541] is applied).

4.1. Serialising Structured Headers into HTTP/1

Given a structured defined in this specification:

1. If the structure is a dictionary, return the result of Serialising a Dictionary {#ser-dictionary}.
2. If the structure is a list, return the result of Serialising a List {#ser-list}.
3. If the structure is a parameterised list, return the result of Serialising a Parameterised List {#ser-param-list}.
4. If the structure is an item, return the result of Serialising an Item {#ser-item}.
5. Otherwise, fail serialisation.

4.1.1. Serialising a Dictionary

Given a dictionary as input:

1. Let output be an empty string.
2. For each member mem of input:
    1. Let name be the result of applying Serialising an Identifier Section 4.1.8 to mem's member-name.
    2. Append name to output.
    3. Append "=" to output.

    4. Let value be the result of applying Serialising an Item Section 4.1.4 to mem's member-value.
    5. Append value to output.
3. Return output.

4.1.2. Serialising a List

Given a list as input:

1. Let output be an empty string.
2. For each member mem of input:
    1. Let value be the result of applying Serialising an Item Section 4.1.4 to mem.
    2. Append value to output.
    3. If more members remain in input:
        1. Append a COMMA to output.
        2. Append a single WS to output.
3. Return output.

4.1.3. Serialising a Parameterised List

Given a parameterised list as input:

1. Let output be an empty string.
2. For each member mem of input:
    1. Let id be the result of applying Serialising an Identifier Section 4.1.8 to mem's identifier.
    2. Append id to output.
    3. For each parameter in mem's parameters:
        1. Let name be the result of applying Serialising an Identifier Section 4.1.8 to parameter's param-name.
        2. Append name to output.

        3. If parameter has a param-value:
            1. Let value be the result of applying Serialising an Item Section 4.1.4 to parameter's param-value.
            2. Append "=" to output.
            3. Append value to output.

3. Return output.

4.1.4. Serialising an Item

Given an item as input:

1. If input is a type other than an integer, float, string or binary content, fail serialisation.
2. Let output be an empty string.
3. If input is an integer, let value be the result of applying Serialising an Integer Section 4.1.5 to input.
4. If input is a float, let value be the result of applying Serialising a Float Section 4.1.6 to input.
5. If input is a string, let value be the result of applying Serialising a String Section 4.1.7 to input.
6. If input is binary content, let value be the result of applying Serialising Binary Content Section 4.1.9 to input.
7. Return output.

4.1.5. Serialising an Integer

Given an integer as input:

1. If input is not an integer in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive, fail serialisation.
2. Let output be an empty string.
3. If input is less than (but not equal to) 0, append "-" to output.
4. Append input's numeric value represented in base 10 using only decimal digits to output.

5. Return output.

4.1.6. Serialising a Float

Given a float as input:

1. If input is not a IEEE 754 double precision number, fail serialisation.
2. Let output be an empty string.
3. If input is less than (but not equal to) 0, append "-" to output.
4. Append input's integer component represented in base 10 using only decimal digits to output; if it is zero, append "0".
5. Append "." to output.
6. Append input's decimal component represented in base 10 using only decimal digits to output; if it is zero, append "0".
7. Return output.

4.1.7. Serialising a String

Given a string as input:

1. If input is not a sequence of characters, or contains characters outside the range allowed by VCHAR, fail serialisation.
2. Let output be an empty string.
3. Append DQUOTE to output.
4. For each character char in input:
    1. If char is "\" or DQUOTE:
        1. Append "\" to output.
    2. Append char to output, using ASCII encoding [RFC0020].
5. Append DQUOTE to output.
6. Return output.

4.1.8. Serialising an Identifier

Given an identifier as input:

1. If input is not a sequence of characters, or contains characters not allowed in Section 3.8, fail serialisation.
2. Let output be an empty string.
3. Append input to output, using ASCII encoding [RFC0020].
4. Return output.

4.1.9. Serialising Binary Content

Given binary content as input:

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

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

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

4.2. Parsing HTTP/1 Header Fields into Structured 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, and header_type, one of "dictionary", "list", "param-list", or "item", return the parsed header value.

1. Discard any leading OWS from input_string.
2. If header_type is "dictionary", let output be the result of Parsing a Dictionary from Text (Section 4.2.1).
3. If header_type is "list", let output be the result of Parsing a List from Text (Section 4.2.2).
4. If header_type is "param-list", let output be the result of Parsing a Parameterised List from Text (Section 4.2.3).
5. Otherwise, let output be the result of Parsing an Item from Text (Section 4.2.5).
6. Discard any leading OWS from input_string.
7. If input_string is not empty, fail parsing.
8. Otherwise, return output.

When generating input_string, parsers MUST combine all instances of the target header field into one comma-separated field-value, as per [RFC7230], Section 3.2.2; this assures that the header is processed correctly.

For Lists, Parameterised Lists and Dictionaries, this has the effect of correctly concatenating all instances of the header field.

Strings can but SHOULD NOT be split across multiple header instances, because comma(s) inserted upon combination will become part of the string output by the parser.

Integers, Floats and Binary Content cannot be split across multiple headers because the inserted commas will cause parsing to fail.

If parsing fails - including when calling another algorithm - the entire header field's value MUST be discarded. This is intentionally strict, to improve interoperability and safety, and specifications referencing this document cannot loosen this requirement.

Note that this has the effect of discarding any header field with non-ASCII characters in input_string.

4.2.1. Parsing a Dictionary from Text

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

1. Let dictionary be an empty, unordered mapping.
2. While input_string is not empty:
    1. Let this_key be the result of running Parse Identifier from Text (Section 4.2.8) with input_string.
    2. If dictionary already contains this_key, fail parsing.
    3. Consume the first character of input_string; if it is not "=", fail parsing.
    4. Let this_value be the result of running Parse Item from Text (Section 4.2.5) with input_string.
    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 the first character of input_string; if it is not COMMA, fail parsing.
    9. Discard any leading OWS from input_string.
    10. If input_string is empty, fail parsing.
3. No structured data has been found; fail parsing.

4.2.2. Parsing a List from Text

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 Text (Section 4.2.5) with input_string.
    2. Append item to items.
    3. Discard any leading OWS from input_string.
    4. If input_string is empty, return items.

    5. Consume the first character of input_string; if it is not COMMA, fail parsing.

    6. Discard any leading OWS from input_string.

    7. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.3. Parsing a Parameterised List from Text

Given an ASCII string input_string, return a list of parameterised identifiers. 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 Parameterised Identifier from Text (Section 4.2.4) with input_string.

    2. Append item to items.

    3. Discard any leading OWS from input_string.

    4. If input_string is empty, return items.

    5. Consume the first character of input_string; if it is not COMMA, fail parsing.

    6. Discard any leading OWS from input_string.

    7. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.4. Parsing a Parameterised Identifier from Text

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

1. Let primary_identifier be the result of Parsing a Identifier from Text (Section 4.2.8) from input_string.

2. Let parameters be an empty, unordered mapping.

3. In a loop:

    1. Discard any leading OWS from 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. Discard any leading OWS from input_string.

    5. let param_name be the result of Parsing a Identifier from Text (Section 4.2.8) from input_string.

    6. If param_name is already present in parameters, fail parsing.

    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 Text (Section 4.2.5) from input_string.

    9. Insert (param_name, param_value) into parameters.

4. Return the tuple (primary_identifier, parameters).

4.2.5. Parsing an Item from Text

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

1. Discard any leading OWS from input_string.

2. If the first character of input_string is a "-" or a DIGIT, process input_string as a number (Section 4.2.6) and return the result.

3. If the first character of input_string is a DQUOTE, process input_string as a string (Section 4.2.7) and return the result.

4. If the first character of input_string is "*", process input_string as binary content (Section 4.2.9) and return the result.

5. Otherwise, fail parsing.

4.2.6. Parsing a Number from Text

NOTE: This algorithm parses both Integers Section 3.5 and Floats Section 3.6, and returns the corresponding structure.

1. Let type be "integer".
2. Let sign be 1.
3. Let input_number be an empty string.
4. If the first character of input_string is "-", remove it from input_string and set sign to -1.
5. If input_string is empty, fail parsing.
6. If the first character of input_string is not a DIGIT, fail parsing.
7. While input_string is not empty:
    1. Let char be the result of removing the first character of input_string.
    2. If char is a DIGIT, append it to input_number.
    3. Else, if type is "integer" and char is ".", append char to input_number and set type to "float".
    4. Otherwise, fail parsing.
    5. If type is "integer" and input_number contains more than 19 characters, fail parsing.
    6. If type is "float" and input_number contains more than 16 characters, fail parsing.
8. If type is "integer":
    1. Parse input_number as an integer and let output_number be the result.
    2. If output_number is outside the range defined in Section 3.5, fail parsing.
9. Otherwise:
    1. If the final character of input_number is ".", fail parsing.

    2. Parse input_number as a float and let output_number be the result.

10. Return the product of output_number and sign.

4.2.7. Parsing a String from Text

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, fail parsing.

3. Discard the first character of input_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 a backslash ("\"):

        1. If input_string is now empty, fail parsing.

        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 "\", fail parsing.

            3. Append next_char to output_string.

    3. Else, if char is DQUOTE, return output_string.

    4. Else, if char is in the range %x00-1f or %x7f (i.e., is not in VCHAR), fail parsing.

    5. Else, append char to output_string.

5. Otherwise, fail parsing.

4.2.8. Parsing an Identifier from Text

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

1. If the first character of input_string is not lcalpha, fail parsing.
2. Let output_string be an empty string.
3. 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.
4. Return output_string.

4.2.9. Parsing Binary Content from Text

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 "*", fail parsing.
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 instance of the character "*". If there is not a "*" character before the end of input_string, fail parsing.
4. Consume the "*" character at the beginning of input_string.
5. If b64_content contains a character not included in ALPHA, DIGIT, "+", "/" and "=", fail parsing.
6. Let binary_content be the result of Base 64 Decoding [RFC4648] b64_content, synthesising padding if necessary (note the requirements about recipient behaviour below).

7. Return binary_content.

As per [RFC4648], Section 3.2, it is RECOMMENDED that parsers reject encoded data that is not properly padded, although this might not be possible in some base64 implementations.

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

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

5. IANA Considerations

This draft has no actions for IANA.

6. Security Considerations

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

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

7. References

7.1. Normative References

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

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

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

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

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

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

7.2. Informative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2008, DOI 10.1109/IEEESTD.2008.4610935, ISBN 978-0-7381-5752-8, August 2008, <http://ieeexplore.ieee.org/document/4610935/>.

See also http://grouper.ieee.org/groups/754/ [6].

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

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

7.3. URIs

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

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

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

[4] https://github.com/httpwg/structured-header-tests

[5] https://github.com/httpwg/wiki/wiki/Structured-Headers

Appendix A. Frequently Asked Questions

A.1. Why not JSON?

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

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

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

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

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

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

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

A.2. Structured Headers don't "fit" my data.

Structured headers intentionally limits the complexity of data structures, to assure that it can be processed in a performant manner with little overhead. This means that work is necessary to fit some data types into them.

Sometimes, this can be achieved by creating limited substructures in values, and/or using more than one header. For example, consider:

```
Example-Thing: name="Widget", cost=89.2, descriptions="foo bar"
Example-Description: foo; url="https://example.net"; context=123,
                     bar; url="https://example.org"; context=456
```

Since the description contains a list of key/value pairs, we use a Parameterised List to represent them, with the identifier for each item in the list used to identify it in the "descriptions" member of the Example-Thing header.

When specifying more than one header, it's important to remember to describe what a processor's behaviour should be when one of the headers is missing.

If you need to fit arbitrarily complex data into a header, Structured Headers is probably a poor fit for your use case.

Appendix B. Changes

RFC Editor: Please remove this section before publication.

B.1. Since draft-ietf-httpbis-header-structure-06

- Add a FAQ.
- Allow non-zero pad bits.
- Explicitly check for integers that violate constraints.

B.2. Since draft-ietf-httpbis-header-structure-05

- Reorganise specification to separate parsing out.
- Allow referencing specs to use ABNF.
- Define serialisation algorithms.

- Refine relationship between ABNF, parsing and serialisation algorithms.

B.3. Since draft-ietf-httpbis-header-structure-04

- Remove identifiers from item.
- Remove most limits on sizes.
- Refine number parsing.

B.4. Since draft-ietf-httpbis-header-structure-03

- Strengthen language around failure handling.

B.5. Since draft-ietf-httpbis-header-structure-02

- Split Numbers into Integers and Floats.
- Define number parsing.
- Tighten up binary parsing and give it an explicit end delimiter.
- Clarify that mappings are unordered.
- Allow zero-length strings.
- Improve string parsing algorithm.
- Improve limits in algorithms.
- Require parsers to combine header fields before processing.
- Throw an error on trailing garbage.

B.6. Since draft-ietf-httpbis-header-structure-01

- Replaced with draft-nottingham-structured-headers.

B.7. 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