

HTTP  
Internet-Draft  
Intended status: Standards Track  
Expires: December 1, 2018

M. Nottingham  
Fastly  
P-H. Kamp  
The Varnish Cache Project  
May 30, 2018

**Structured Headers for HTTP**  
**draft-ietf-httpbis-header-structure-05**

Abstract

This document describes a set of data types and parsing 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](mailto: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 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 December 1, 2018.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

|                             |  |                    |
|-----------------------------|--|--------------------|
| <a href="#">1.</a>          | Introduction . . . . .   | <a href="#">3</a>  |
| <a href="#">1.1.</a>        | Notational Conventions . . . . .                                       | <a href="#">4</a>  |
| <a href="#">2.</a>          | Defining New Structured Headers . . . . .                              | <a href="#">4</a>  |
| <a href="#">3.</a>          | Parsing Textual Header Fields . . . . .                                | <a href="#">6</a>  |
| <a href="#">4.</a>          | Structured Header Data Types . . . . .                                 | <a href="#">7</a>  |
| <a href="#">4.1.</a>        | Dictionaries . . . . .   | <a href="#">7</a>  |
| <a href="#">4.2.</a>        | Lists . . . . .  | <a href="#">8</a>  |
| <a href="#">4.3.</a>        | Parameterised Lists . . . . .  | <a href="#">9</a>  |
| <a href="#">4.4.</a>        | Items . . . . .  | <a href="#">11</a> |
| <a href="#">4.5.</a>        | Integers . . . . .   | <a href="#">12</a> |
| <a href="#">4.6.</a>        | Floats . . . . .   | <a href="#">13</a> |
| <a href="#">4.7.</a>        | Strings . . . . .  | <a href="#">14</a> |
| <a href="#">4.8.</a>        | Identifiers . . . . .  | <a href="#">15</a> |
| <a href="#">4.9.</a>        | Binary Content . . . . .   | <a href="#">16</a> |
| <a href="#">5.</a>          | IANA Considerations . . . . .  | <a href="#">17</a> |
| <a href="#">6.</a>          | Security Considerations . . . . .                                      | <a href="#">17</a> |
| <a href="#">7.</a>          | References . . . . .   | <a href="#">17</a> |
| <a href="#">7.1.</a>        | Normative References . . . . .   | <a href="#">18</a> |
| <a href="#">7.2.</a>        | Informative References . . . . .                                       | <a href="#">18</a> |
| <a href="#">7.3.</a>        | URIs . . . . .   | <a href="#">19</a> |
| <a href="#">Appendix A.</a> | Changes . . . . .  | <a href="#">19</a> |
| <a href="#">A.1.</a>        | Since <a href="#">draft-ietf-httpbis-header-structure-04</a> . . . . . | <a href="#">19</a> |
| <a href="#">A.2.</a>        | Since <a href="#">draft-ietf-httpbis-header-structure-03</a> . . . . . | <a href="#">19</a> |
| <a href="#">A.3.</a>        | Since <a href="#">draft-ietf-httpbis-header-structure-02</a> . . . . . | <a href="#">19</a> |

Nottingham & Kamp

Expires December 1, 2018

[Page 2]

|                      |  |           |                    |
|----------------------|--|-----------|--------------------|
| <a href="#">A.4.</a> | Since <a href="#">draft-ietf-httpbis-header-structure-01</a> | . . . . . | <a href="#">20</a> |
| <a href="#">A.5.</a> | Since <a href="#">draft-ietf-httpbis-header-structure-00</a> | . . . . . | <a href="#">20</a> |
| Authors' Addresses   | . . . . .  |           | <a href="#">20</a> |

## **[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 for it 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 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 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 4](#) defines a number of abstract data types that can be used in Structured Headers. Dictionaries and lists are only usable at the "top" level, while the remaining types can be specified appear at the top level or inside those structures.

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](#)].

This document uses algorithms to specify normative parsing behaviours, and ABNF to illustrate the on-wire format expected. Implementations MUST follow the normative algorithms, but MAY vary in implementation so as the behaviours are indistinguishable from specified behaviour. If there is disagreement between the algorithms and ABNF, the specified algorithms take precedence.

## **2. Defining New Structured Headers**

A HTTP header that uses the structures in this specification 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.

A header field definition cannot relax or otherwise modify the requirements of this specification, or change the nature of its data structures; doing so would preclude handling by generic software.

However, header field authors are encouraged to clearly state additional constraints upon the syntax, as well as the consequences when those constraints are violated. When Structured Headers parsing fails, the header is discarded (see [Section 3](#)); in most situations, header-specific constraints should do likewise.

Such constraints could include additional structure inside those defined here (e.g., a list of URLs [[RFC3986](#)] inside a string).

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).

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.

Note that specifications using Structured Headers do not re-specify its ABNF or parsing algorithms; instead, they should be specified in terms of its abstract data structures.

Also, empty header field values are not allowed, and therefore parsing for them will fail.





### 3. Parsing Textual Header Fields

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 4.1.1](#)).
3. If `header_type` is "list", let output be the result of Parsing a List from Text ([Section 4.2.1](#)).
4. If `header_type` is "param-list", let output be the result of Parsing a Parameterised List from Text ([Section 4.3.1](#)).
5. Otherwise, let output be the result of Parsing an Item from Text ([Section 4.4.1](#)).
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. 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. Dictionaries**

Dictionaries are unordered maps of key-value pairs, where the keys are identifiers ([Section 4.8](#)) and the values are items ([Section 4.4](#)). There can be one or more 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. Duplicate keys MUST cause parsing to fail.

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

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

```
Example-DictHeader: 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.

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

#### **4.1.1. 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 4.8.1](#)) with `input_string`.
  2. If dictionary already contains `this_key`, fail parsing.
  3. Consume a "=" from `input_string`; if none is present, fail parsing.
  4. Let `this_value` be the result of running Parse Item from Text ([Section 4.4.1](#)) 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 a COMMA from `input_string`; if no comma is present, 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.](#) Lists**

Lists are arrays of items ([Section 4.4](#)) with one or more members.

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

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

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."
```

Parsers MUST support lists containing at least 1024 members.



#### **4.2.1. 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 4.4.1](#)) 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 a COMMA from `input_string`; if no comma is present, 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.

#### **4.3. Parameterised Lists**

Parameterised Lists are arrays of a parameterised identifiers.

A parameterised identifier is an identifier ([Section 4.8](#)) with an optional set of parameters, each parameter having a identifier and an optional value that is an item ([Section 4.4](#)). Ordering between parameters is not significant, and duplicate parameters MUST cause parsing to fail.

In the textual HTTP serialisation, each parameterised identifier is separated by a comma and optional whitespace. Parameters are delimited from each other using semicolons (";"), and equals ("=") delimits the parameter name from its value.

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

For example,

Example-ParamListHeader: abc\_123;a=1;b=2; c, def\_456, ghi;q="19";r=foo





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

#### **4.3.1. 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 4.3.2](#)) 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 a COMMA from `input_string`; if no comma is present, 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.

#### **4.3.2. 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 4.8.1](#)) 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 4.8.1](#)) 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 4.4.1](#)) from `input_string`.
9. Insert (`param_name`, `param_value`) into `parameters`.
4. Return the tuple (`primary_identifier`, `parameters`).

#### **[4.4.](#) Items**

An item is can be a integer ([Section 4.5](#)), float ([Section 4.6](#)), string ([Section 4.7](#)), or binary content ([Section 4.9](#)).

`item` = integer / float / string / binary

##### **[4.4.1.](#) 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 4.5.1](#)) and return the result.
3. If the first character of `input_string` is a DQUOTE, process `input_string` as a string ([Section 4.7.1](#)) and return the result.
4. If the first character of `input_string` is "\*", process `input_string` as binary content ([Section 4.9.1](#)) and return the result.



5. Otherwise, fail parsing.

#### **4.5. Integers**

Abstractly, 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).

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

Parsers that encounter an integer outside the range defined above MUST fail parsing. Therefore, the value "9223372036854775808" would be invalid. Likewise, values that do not conform to the ABNF above are invalid, and MUST fail parsing.

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

```
Example-IntegerHeader: 42
```

##### **4.5.1. Parsing a Number from Text**

NOTE: This algorithm parses both Integers and Floats [Section 4.6](#), 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", parse input\_number as an integer and let output\_number be the result.
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.

#### **4.6. Floats**

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

The textual HTTP serialisation of floats allows a maximum of fifteen digits between the integer and fractional part, with at least one required on each side, along with an optional "-" indicating negative numbers.

```
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 )
```

Values that do not conform to the ABNF above are invalid, and MUST fail parsing.





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

Example-FloatHeader: 4.5

See [Section 4.5.1](#) for the parsing algorithm for floats.

#### **[4.7.](#) Strings**

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

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

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

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 4.9](#)) SHOULD be specified, along with a character encoding (preferably, UTF-8).

Parsers MUST support strings with at least 1024 characters.

##### **[4.7.1.](#) 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, append `char` to `output_string`.
5. Otherwise, fail parsing.

#### **4.8. 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.

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

Note that identifiers can only contain lowercase letters.

##### **4.8.1. Parsing a 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.

#### **4.9. Binary Content**

Arbitrary binary content can be conveyed in Structured Headers.

The textual HTTP serialisation encodes the data using Base 64 Encoding [\[RFC4648\]](#), [Section 4](#), and surrounds it with a pair of asterisks ("\*") to delimit from other content.

The encoded data is required to be padded with "=", 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 with some base64 implementations.

Likewise, encoded data is required to have pad bits set to zero, as per [\[RFC4648\]](#), [Section 3.5](#). It is RECOMMENDED that parsers fail on encoded data that has non-zero pad bits, although this might not be possible with some base64 implementations.

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.

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

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

```
Example-BinaryHeader: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==*
```



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

#### **[4.9.1.](#) 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. Let `binary_content` be the result of Base 64 Decoding [[RFC4648](#)] `b64_content`, synthesising padding if necessary (note the requirements about recipient behaviour in [Section 4.9](#)).
6. Return `binary_content`.

## **[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].
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, [RFC 3986](#), DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [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>
- [4] <https://github.com/httpwg/structured-header-tests>
- [5] <https://github.com/httpwg/wiki/wiki/Structured-Headers>

## **Appendix A. Changes**

### **A.1. Since [draft-ietf-httpbis-header-structure-04](#)**

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

### **A.2. Since [draft-ietf-httpbis-header-structure-03](#)**

- o Strengthen language around failure handling.

### **A.3. Since [draft-ietf-httpbis-header-structure-02](#)**

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



- o Throw an error on trailing garbage.

**A.4. Since [draft-ietf-httpbis-header-structure-01](#)**

- o Replaced with [draft-nottingham-structured-headers](#).

**A.5. Since [draft-ietf-httpbis-header-structure-00](#)**

- o Added signed 64bit integer type.
- o Drop UTF8, and settle on [BCP137](#) ::EmbeddedUnicodeChar for h1-unicode-string.
- o Change h1\_blob delimiter to ":" since '"' is valid t\_char

Authors' Addresses

Mark Nottingham  
Fastly

Email: [mnot@mnot.net](mailto:mnot@mnot.net)  
URI: <https://www.mnot.net/>

Poul-Henning Kamp  
The Varnish Cache Project

Email: [phk@varnish-cache.org](mailto:phk@varnish-cache.org)

