

HTTP
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2020

M. Nottingham
Fastly
P-H. Kamp
The Varnish Cache Project
July 8, 2019

Structured Headers for HTTP
draft-ietf-httpbis-header-structure-11

Abstract

This document describes a set of data types and associated algorithms that are intended to make it easier and safer to define and handle HTTP header fields. It is intended for use by specifications of new HTTP header fields that wish to use a common syntax that is more restrictive than traditional HTTP field values.

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 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-Draft

Structured Headers for HTTP

July 2019

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 January 9, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](https://trustee.ietf.org/bcp78) 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

1.	Introduction	3
1.1.	Intentionally Strict Processing	4
1.2.	Notational Conventions	4
2.	Defining New Structured Headers	5
3.	Structured Header Data Types	6
3.1.	Lists	6
3.2.	Dictionaries	7
3.3.	Items	8
3.4.	Integers	8
3.5.	Floats	9
3.6.	Strings	9
3.7.	Tokens	10
3.8.	Byte Sequences	10
3.9.	Booleans	11
4.	Working With Structured Headers in Textual HTTP Headers	11
4.1.	Serializing Structured Headers	11
4.2.	Parsing Header Fields into Structured Headers	17
5.	IANA Considerations	26
6.	Security Considerations	26

7.	References	26
7.1.	Normative References	26
7.2.	Informative References	27
7.3.	URIs	28
Appendix A.	Acknowledgements	28

Appendix B.	Frequently Asked Questions	28
B.1.	Why not JSON?	28
B.2.	Structured Headers don't "fit" my data.	29
Appendix C.	Implementation Notes	29
Appendix D.	Changes	30
D.1.	Since draft-ietf-httpbis-header-structure-10	30
D.2.	Since draft-ietf-httpbis-header-structure-09	30
D.3.	Since draft-ietf-httpbis-header-structure-08	31
D.4.	Since draft-ietf-httpbis-header-structure-07	31
D.5.	Since draft-ietf-httpbis-header-structure-06	32
D.6.	Since draft-ietf-httpbis-header-structure-05	32
D.7.	Since draft-ietf-httpbis-header-structure-04	32
D.8.	Since draft-ietf-httpbis-header-structure-03	32
D.9.	Since draft-ietf-httpbis-header-structure-02	32
D.10.	Since draft-ietf-httpbis-header-structure-01	33
D.11.	Since draft-ietf-httpbis-header-structure-00	33
Authors' Addresses		33

[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 serializers 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 definitions of new 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 [\[RFC7230\]](#) 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.

[Section 2](#) describes how to specify a Structured Header.

[Section 3](#) defines a number of abstract data types that can be used in Structured Headers. Those abstract types can be serialized into and parsed from textual HTTP headers using the algorithms described in [Section 4](#).

[1.1](#). Intentionally Strict Processing

This specification intentionally defines strict parsing and serialisation behaviours using step-by-step algorithms; the only error handling defined is to fail the operation altogether.

It is designed to encourage faithful implementation and therefore good interoperability. Therefore, an implementation that tried to be "helpful" by being more tolerant of input would make interoperability worse, since that would create pressure on other implementations to implement similar (but likely subtly different) workarounds.

In other words, strict processing is an intentional feature of this specification; it allows non-conformant input to be discovered and corrected by the producer early, and avoids both interoperability and security issues that might otherwise result.

Note that as a result of this strictness, if a header field is appended to by multiple parties (e.g., intermediaries, or different components in the sender), an error in one party's value is likely to cause the entire header field to fail parsing.

[1.2.](#) 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 algorithms to specify parsing and serialisation behaviours, and the Augmented Backus-Naur Form (ABNF) notation of [[RFC5234](#)] to illustrate expected syntax in textual HTTP header fields. In doing so, uses the VCHAR, SP, DIGIT, ALPHA and DQUOTE rules from [[RFC5234](#)]. It also includes the OWS rule from [[RFC7230](#)].

When parsing from textual HTTP header fields, 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. In some places, the algorithms are "greedy" with whitespace, but this should not affect conformance.

For serialisation to textual header fields, 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:

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

- o 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 this specification because doing so would preclude handling by generic software; they can only add additional constraints (for example, on the numeric range of integers and floats, the format of strings and tokens, or the number of items in a list). Likewise, header field definitions should use Structured Headers for the entire header field value, not a portion thereof.

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.

For example,

42. 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 name is "foo", and whose value is an integer ([RFCxxxx], Section Y.Y), indicating the number of foos in the message.

* Exactly one member whose name is "barUrl", and whose value is a string ([RFCxxxx], Section Y.Y), conveying the Bar URL 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.

"barUrl" contains a URI-reference ([\[RFC3986\], Section 4.1](#)).

If barURL is not a valid URI-reference, it MUST be ignored. If barURL is a relative reference ([\[RFC3986\], Section 4.2](#)), it MUST be resolved ([\[RFC3986\], Section 5](#)) before being used.

[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.

[3.1.](#) Lists

Lists are arrays of zero or more members, each of which can be an item ([Section 3.3](#)) or an inner list (an array of zero or more items).

Each member of the top-level list can also have associated parameters – an ordered map of key-value pairs where the keys are short, textual strings and the values are items ([Section 3.3](#)). There can be zero or more parameters on a member, and their keys are required to be unique within that scope.

The ABNF for lists is:

```
sh-list      = list-member *( OWS "," OWS list-member )
list-member  = ( sh-item / inner-list ) *parameter
inner-list   = "(" OWS [ sh-item *( SP sh-item ) OWS ] ")"
parameter    = OWS ";" OWS param-name [ "=" param-value ]
param-name   = key
key          = lcalpha *( lcalpha / DIGIT / "_" / "-" )
```

```
lcalpha      = %x61-7A ; a-z
param-value  = sh-item
```

In textual HTTP headers, 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."
```

In textual HTTP headers, inner lists are denoted by surrounding parenthesis, and have their values delimited by a single space. A header field whose value is defined as a list of lists of strings could look like:

```
Example-StrListListHeader: ("foo" "bar"), ("baz"), ("bat" "one"), ()
```

Note that the last member in this example is an empty inner list.

In textual HTTP headers, members' parameters are separated from the member and each other by semicolons. For example:

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

Parsers MUST support lists containing at least 1024 members, support members with at least 256 parameters, support inner-lists containing at least 256 members, and support parameter keys with at least 64 characters.

Header specifications can constrain the types of individual list values (including that of individual inner-list members and parameters) if necessary.

[3.2.](#) Dictionaries

Dictionaries are ordered maps of name-value pairs, where the names are short, textual strings and the values are items ([Section 3.3](#)) or arrays of items. There can be zero or more members, and their names are required to be unique within the scope of the dictionary they occur within.

Implementations MUST provide access to dictionaries both by index and

by name. Specifications MAY use either means of accessing the members.

The ABNF for dictionaries in textual HTTP headers is:

```
sh-dictionary = dict-member *( OWS "," OWS dict-member )
dict-member   = member-name "=" member-value
member-name   = key
member-value  = sh-item / inner-list
```

In textual HTTP headers, members are separated by a comma with optional whitespace, while names and values are separated by "=" (without whitespace). For example:

```
Example-DictHeader: en="Applepie", da=*w4ZibGV0w6ZydGU=*
```

A dictionary with a member whose value is an inner-list of tokens:

```
Example-DictListHeader: rating=1.5, feelings=(joy sadness)
```

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

Parsers MUST support dictionaries containing at least 1024 name/value pairs, and names with at least 64 characters.

[3.3.](#) Items

An item can be an integer ([Section 3.4](#)), float ([Section 3.5](#)), string ([Section 3.6](#)), token ([Section 3.7](#)), byte sequence ([Section 3.8](#)), or Boolean ([Section 3.9](#)).

The ABNF for items in textual HTTP headers is:

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

[3.4.](#) Integers

Integers have a range of -999,999,999,999,999 to 999,999,999,999,999 inclusive (i.e., up to fifteen digits, signed), for IEEE 754 compatibility ([\[IEEE754\]](#)).

The ABNF for integers in textual HTTP headers is:

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

For example:

```
Example-IntegerHeader: 42
```

Note that commas in integers are used in this section's prose only for readability; they are not valid in the wire format.

[3.5.](#) Floats

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

The ABNF for floats in textual HTTP headers 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
```

[3.6.](#) 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 in textual HTTP headers is:

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

In textual HTTP 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, a byte sequence ([Section 3.8](#)) SHOULD be specified, along with a character encoding (preferably UTF-8).

Parsers MUST support strings with at least 1024 characters.

[3.7.](#) Tokens

Tokens are short textual words; their abstract model is identical to their expression in the textual HTTP serialisation.

The ABNF for tokens in textual HTTP headers is:

```
sh-token = ALPHA
          *( ALPHA / DIGIT / "_" / "-" / "." / ":" / "%"
            / "*" / "/" )
```

Parsers MUST support tokens with at least 512 characters.

Note that a Structured Header token is not the same as the "token"

ABNF rule defined in [[RFC7230](#)].

[3.8.](#) Byte Sequences

Byte sequences can be conveyed in Structured Headers.

The ABNF for a byte sequence in textual HTTP headers is:

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

In textual HTTP headers, a byte sequence is delimited with asterisks and encoded using base64 ([\[RFC4648\]](#), [Section 4](#)). For example:

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

Parsers MUST support byte sequences with at least 16384 octets after decoding.

[3.9.](#) Booleans

Boolean values can be conveyed in Structured Headers.

The ABNF for a Boolean in textual HTTP headers is:

```
sh-boolean = "?" boolean
boolean    = "0" / "1"
```

In textual HTTP headers, a boolean is indicated with a leading "?" character. For example:

```
Example-BoolHdr: ?1
```

[4.](#) Working With Structured Headers in Textual HTTP Headers

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

[4.1.](#) Serializing Structured Headers

Given a structure defined in this specification:

1. If the structure is a dictionary or list and its value is empty (i.e., it has no members), do not send the serialize field at all (i.e., omit both the field-name and field-value).
2. If the structure is a dictionary, let `output_string` be the result of Serializing a Dictionary ([Section 4.1.2](#)).
3. Else if the structure is a list, let `output_string` be the result of Serializing a List [Section 4.1.1](#).
4. Else if the structure is an item, let `output_string` be the result of Serializing an Item ([Section 4.1.3](#)).

5. Else, fail serialisation.
6. Return `output_string` converted into an array of bytes, using ASCII encoding [[RFC0020](#)].

[4.1.1](#). Serializing a List

Given a list of (member, parameters) as `input_list`:

1. Let output be an empty string.
2. For each (member, parameters) of `input_list`:
 1. If member is an array, let `mem_value` be the result of applying Serialising an Inner List ([Section 4.1.1.1](#)) to member.
 2. Otherwise, let `mem_value` be the result of applying Serializing an Item ([Section 4.1.3](#)) to member.
 3. Append `mem_value` to output.
 4. For each parameter in parameters:
 1. Append ";" to output.

2. Let name be the result of applying Serializing a Key ([Section 4.1.1.2](#)) to parameter's param-name.
 3. Append name to output.
 4. If parameter has a param-value:
 1. Let value be the result of applying Serializing an Item ([Section 4.1.3](#)) to parameter's param-value.
 2. Append "=" to output.
 3. Append value to output.
 5. If more members remain in input_plist:
 1. Append a COMMA to output.
 2. Append a single WS to output.
3. Return output.

[4.1.1.1](#). Serialising an Inner List

Given an array inner_list:

1. Let output be the string "(".
2. For each member mem of inner_list:
 1. Let value be the result of applying Serializing an Item ([Section 4.1.3](#)) to mem.
 2. Append value to output.
 3. If inner_list is not empty, append a single WS to output.
3. Append ")" to output.
4. Return output.

[4.1.1.2](#). Serializing a Key

Given a key as `input_key`:

1. If `input_key` is not a sequence of characters, or contains characters not allowed in the ABNF for key, fail serialisation.
2. Let output be an empty string.
3. Append `input_key` to output.
4. Return output.

[4.1.2](#). Serializing a Dictionary

Given a dictionary as `input_dictionary`:

1. Let output be an empty string.
2. For each member `mem` of `input_dictionary`:
 1. Let `name` be the result of applying Serializing a Key ([Section 4.1.1.2](#)) to `mem`'s member-name.
 2. Append `name` to output.
 3. Append "=" to output.

4. If `mem` is an array, let `value` be the result of applying Serialising an Inner List ([Section 4.1.1.1](#)) to `mem`.
5. Otherwise, let `value` be the result of applying Serializing an Item ([Section 4.1.3](#)) to `mem`.
6. Append `value` to output.
7. If more members remain in `input_dictionary`:
 1. Append a COMMA to output.

2. Append a single WS to output.
3. Return output.

[4.1.3.](#) Serializing an Item

Given an item as `input_item`:

1. If `input_item` is an integer, return the result of applying Serializing an Integer ([Section 4.1.4](#)) to `input_item`.
2. If `input_item` is a float, return the result of applying Serializing a Float ([Section 4.1.5](#)) to `input_item`.
3. If `input_item` is a string, return the result of applying Serializing a String ([Section 4.1.6](#)) to `input_item`.
4. If `input_item` is a token, return the result of Serializing a Token ([Section 4.1.7](#)) to `input_item`.
5. If `input_item` is a Boolean, return the result of applying Serializing a Boolean ([Section 4.1.9](#)) to `input_item`.
6. If `input_item` is a byte sequence, return the result of applying Serializing a Byte Sequence ([Section 4.1.8](#)) to `input_item`.
7. Otherwise, fail serialisation.

[4.1.4.](#) Serializing an Integer

Given an integer as `input_integer`:

1. If `input_integer` is not an integer in the range of -999,999,999,999,999 to 999,999,999,999,999 inclusive, fail serialisation.

2. Let output be an empty string.
3. If `input_integer` is less than (but not equal to) 0, append "-" to output.

4. Append `input_integer`'s numeric value represented in base 10 using only decimal digits to output.
5. Return output.

[4.1.5. Serializing a Float](#)

Given a float as `input_float`:

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

[4.1.6. Serializing a String](#)

Given a string as `input_string`:

1. If `input_string` is not a sequence of characters, or contains characters outside the range allowed by VCHAR or SP, fail serialisation.
2. Let output be an empty string.
3. Append DQUOTE to output.
4. For each character `char` in `input_string`:
 1. If `char` is "\" or DQUOTE:

1. Append "\" to output.
2. Append char to output.
5. Append DQUOTE to output.
6. Return output.

[4.1.7.](#) Serializing a Token

Given a token as `input_token`:

1. If `input_token` is not a sequence of characters, or contains characters not allowed in [Section 3.7](#), fail serialisation.
2. Let output be an empty string.
3. Append `input_token` to output.
4. Return output.

[4.1.8.](#) Serializing a Byte Sequence

Given a byte sequence as `input_bytes`:

1. If `input_bytes` 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_bytes` 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.

[4.1.9.](#) Serializing a Boolean

Given a Boolean as `input_boolean`:

1. If `input_boolean` is not a boolean, fail serialisation.
2. Let `output` be an empty string.
3. Append "?" to `output`.
4. If `input_boolean` is true, append "1" to `output`.
5. If `input_boolean` is false, append "0" to `output`.
6. Return `output`.

[4.2.](#) Parsing Header Fields into Structured Headers

When a receiving implementation parses textual HTTP header fields 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 array of bytes `input_bytes` that represents the chosen header's field-value (which is an empty string if that header is not present), and `header_type` (one of "dictionary", "list", or "item"), return the parsed header value.

1. Convert `input_bytes` into an ASCII string `input_string`; if conversion fails, fail parsing.
2. Discard any leading OWS from `input_string`.
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 "dictionary", let `output` be the result of Parsing a Dictionary from Text ([Section 4.2.2](#)).

5. If header_type is "item", let output be the result of Parsing an Item from Text ([Section 4.2.4](#)).
6. Discard any leading OWS from input_string.
7. If input_string is not empty, fail parsing.
8. Otherwise, return output.

When generating input_bytes, 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 and Dictionaries, this has the effect of correctly concatenating all instances of the header field, as long as individual individual members of the top-level data structure are not split across multiple header instances.

Strings split across multiple header instances will have unpredictable results, because comma(s) and whitespace inserted upon combination will become part of the string output by the parser. Since concatenation might be done by an upstream intermediary, the results are not under the control of the serializer or the parser.

Tokens, Integers, Floats and Byte Sequences 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 are not allowed to loosen this requirement.

[4.2.1](#). Parsing a List from Text

Given an ASCII string input_string, return an array of (member, parameters). input_string is modified to remove the parsed value.

1. Let members be an empty array.
2. While input_string is not empty:

1. Let member be the result of running Parsing a Parameterized Member from Text ([Section 4.2.1.1](#)) with input_string.
2. Append member to members.
3. Discard any leading OWS from input_string.
4. If input_string is empty, return members.
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, there is a trailing comma; fail parsing.
3. No structured data has been found; return members (which is empty).

[4.2.1.1](#). Parsing a Parameterized Member from Text

Given an ASCII string input_string, return an token with an ordered map of parameters. input_string is modified to remove the parsed value.

1. If the first character of input_string is "(", let member be the result of running Parsing an Inner List ([Section 4.2.1.2](#)) with input_string.
2. Else, let member be the result of running Parsing an Item ([Section 4.2.4](#)) with input_string.
3. Let parameters be an empty, ordered map.
4. 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 key from Text ([Section 4.2.3](#)) from `input_string`.
6. If `param_name` is already present in `parameters`, there is a duplicate; 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 4.2.4](#)) from `input_string`.

9. Append key `param_name` with value `param_value` to `parameters`.
5. Return the tuple (`member`, `parameters`).

[4.2.1.2](#). Parsing an Inner List

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

1. Consume the first character of `input_string`; if it is not "(", fail parsing.
2. Let `inner_list` be an empty array.
3. While `input_string` is not empty:
 1. Discard any leading OWS from `input_string`.
 2. If the first character of `input_string` is ")":

1. Consume the first character of `input_string`.
 2. Return `inner_list`.
 3. Let `item` be the result of running Parsing an Item from Text ([Section 4.2.4](#)) with `input_string`.
 4. Append `item` to `inner_list`.
 5. If the first character of `input_string` is not SP or `)`, fail parsing.
4. The end of the inner list was not found; fail parsing.

[4.2.2](#). Parsing a Dictionary from Text

Given an ASCII string `input_string`, return an ordered map of (key, item). `input_string` is modified to remove the parsed value.

1. Let `dictionary` be an empty, ordered map.
2. While `input_string` is not empty:
 1. Let `this_key` be the result of running Parsing a Key from Text ([Section 4.2.3](#)) with `input_string`.
 2. If `dictionary` already contains the name `this_key`, there is a duplicate; fail parsing.

3. Consume the first character of `input_string`; if it is not `=`, fail parsing.
4. If the first character of `input_string` is `(`, let `this_value` be the result of running Parsing an Inner List ([Section 4.2.1.2](#)) with `input_string`.
5. Else, let `this_value` be the result of running Parsing an Item ([Section 4.2.4](#)) with `input_string`.
6. Add name `this_key` with value `this_value` to `dictionary`.

7. Discard any leading OWS from `input_string`.
 8. If `input_string` is empty, return dictionary.
 9. Consume the first character of `input_string`; if it is not COMMA, fail parsing.
 10. Discard any leading OWS from `input_string`.
 11. If `input_string` is empty, there is a trailing comma; fail parsing.
3. No structured data has been found; return dictionary (which is empty).

[4.2.3.](#) Parsing a Key from Text

Given an ASCII string `input_string`, return a key. `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`.

[4.2.4.](#) Parsing an Item from Text

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

1. If the first character of `input_string` is a "-" or a DIGIT, process `input_string` as a number ([Section 4.2.5](#)) and return the result.
2. If the first character of `input_string` is a DQUOTE, process `input_string` as a string ([Section 4.2.6](#)) and return the result.
3. If the first character of `input_string` is "*", process `input_string` as a byte sequence ([Section 4.2.8](#)) and return the result.
4. If the first character of `input_string` is "?", process `input_string` as a Boolean ([Section 4.2.9](#)) and return the result.
5. If the first character of `input_string` is an ALPHA, process `input_string` as a token ([Section 4.2.7](#)) and return the result.
6. Otherwise, the item type is unrecognized; fail parsing.

[4.2.5](#). Parsing a Number from Text

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

NOTE: This algorithm parses both Integers [Section 3.4](#) and Floats [Section 3.5](#), 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 "-", consume it and set `sign` to -1.
5. If `input_string` is empty, there is an empty integer; 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 consuming 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, prepend `char` to `input_string`, and exit the loop.
 5. If `type` is "integer" and `input_number` contains more than 15 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 product of the result and sign.
 2. If `output_number` is outside the range defined in [Section 3.4](#), 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 product of the result and sign.
10. Return `output_number`.

[4.2.6](#). 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 consuming 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 consuming 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` or `SP`), fail parsing.
 5. Else, append `char` to `output_string`.
5. Reached the end of `input_string` without finding a closing `DQUOTE`; fail parsing.

[4.2.7.](#) Parsing a Token from Text

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

1. If the first character of `input_string` is not `ALPHA`, fail parsing.
2. Let `output_string` be an empty string.
3. While `input_string` is not empty:

1. Let char be the result of consuming the first character of input_string.
2. If char is not one of ALPHA, DIGIT, "_", "-", ".", ":", "%", "*" or "/":

1. Prepend char to input_string.
2. Return output_string.
3. Append char to output_string.
4. Return output_string.

[4.2.8](#). Parsing a Byte Sequence from Text

Given an ASCII string input_string, return a byte sequence. 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. If there is not a "*" character before the end of input_string, fail parsing.
4. Let b64_content be the result of consuming content of input_string up to but not including the first instance of the character "*".
5. Consume the "*" character at the beginning of input_string.
6. If b64_content contains a character not included in ALPHA, DIGIT, "+", "/" and "=", fail parsing.
7. Let binary_content be the result of Base 64 Decoding [[RFC4648](#)] b64_content, synthesizing padding if necessary (note the requirements about recipient behaviour below).
8. Return binary_content.

Because some implementations of base64 do not allow reject of encoded data that is not properly "=" padded (see [\[RFC4648\], Section 3.2](#)), parsers SHOULD NOT fail when it is not present, unless they cannot be configured to do so.

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 do so.

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.

[4.2.9](#). Parsing a Boolean from Text

Given an ASCII string `input_string`, return a Boolean. `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. If the first character of `input_string` matches "1", discard the first character, and return true.
4. If the first character of `input_string` matches "0", discard the first character, and return false.
5. No value has matched; fail parsing.

[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 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 Header. 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>

[6] <https://github.com/httpwg/structured-header-tests>

Appendix A. Acknowledgements

Many thanks to Matthew Kerwin for his detailed feedback and careful consideration during the development of this specification.

Appendix B. Frequently Asked Questions

B.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. In short, if it looks like JSON, people will be

tempted to use a JSON parser / serialiser on header fields.

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 serializer.

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

[B.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 an array of key/value pairs, we use a List to represent them, with the token for each item in the array 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 C.](#) Implementation Notes

A generic implementation of this specification should expose the top-level parse ([Section 4.2](#)) and serialize ([Section 4.1](#)) functions. They need not be functions; for example, it could be implemented as an object, with methods for each of the different top-level types.

For interoperability, it's important that generic implementations be complete and follow the algorithms closely; see [Section 1.1](#). To aid this, a common test suite is being maintained by the community; see <https://github.com/httpwg/structured-header-tests> [7].

Implementers should note that dictionaries and parameters are order-preserving maps. Some headers may not convey meaning in the ordering of these data types, but it should still be exposed so that applications which need to use it will have it available.

Likewise, implementations should note that it's important to preserve the distinction between tokens and strings. While most programming languages have native types that map to the other types well, it may be necessary to create a wrapper "token" object or use a parameter on functions to assure that these types remain separate.

[Appendix D](#). Changes

RFC Editor: Please remove this section before publication.

[D.1](#). Since [draft-ietf-httpbis-header-structure-10](#)

- o Update abstract (#799).
- o Input and output are now arrays of bytes (#662).
- o Implementations need to preserve difference between token and string (#790).
- o Allow empty dictionaries and lists (#781).
- o Change parameterized lists to have primary items (#797).
- o Allow inner lists in both dictionaries and lists; removes lists of lists (#816).
- o Subsume Parameterised Lists into Lists (#839).

[D.2](#). Since [draft-ietf-httpbis-header-structure-09](#)

- o Changed Boolean from T/F to 1/0 (#784).
- o Parameters are now ordered maps (#765).
- o Clamp integers to 15 digits (#737).

D.3. Since [draft-ietf-httpbis-header-structure-08](#)

- o Disallow whitespace before items properly (#703).
- o Created "key" for use in dictionaries and parameters, rather than relying on identifier (#702). Identifiers have a separate minimum supported size.
- o Expanded the range of special characters allowed in identifier to include all of ALPHA, ".", ":", and "%" (#702).
- o Use "?" instead of "!" to indicate a Boolean (#719).
- o Added "Intentionally Strict Processing" (#684).
- o Gave better names for referring specs to use in Parameterised Lists (#720).
- o Added Lists of Lists (#721).
- o Rename Identifier to Token (#725).
- o Add implementation guidance (#727).

D.4. Since [draft-ietf-httpbis-header-structure-07](#)

- o Make Dictionaries ordered mappings (#659).
- o Changed "binary content" to "byte sequence" to align with Infra specification (#671).
- o Changed "mapping" to "map" for #671.
- o Don't fail if byte sequences aren't "=" padded (#658).
- o Add Booleans (#683).
- o Allow identifiers in items again (#629).
- o Disallowed whitespace before items (#703).
- o Explain the consequences of splitting a string across multiple headers (#686).

[D.5.](#) Since [draft-ietf-httpbis-header-structure-06](#)

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

[D.6.](#) Since [draft-ietf-httpbis-header-structure-05](#)

- o Reorganise specification to separate parsing out.
- o Allow referencing specs to use ABNF.
- o Define serialisation algorithms.
- o Refine relationship between ABNF, parsing and serialisation algorithms.

[D.7.](#) Since [draft-ietf-httpbis-header-structure-04](#)

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

[D.8.](#) Since [draft-ietf-httpbis-header-structure-03](#)

- o Strengthen language around failure handling.

[D.9.](#) 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.

D.10. Since [draft-ietf-httpbis-header-structure-01](#)

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

D.11. 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
URI: <https://www.mnot.net/>

Poul-Henning Kamp
The Varnish Cache Project

Email: phk@varnish-cache.org

