Structured Headers for HTTP
draft-ietf-httpbis-header-structure-14

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/.
1. Introduction

Specifying the syntax of new HTTP header fields is an onerous task; even with the guidance in Section 8.3.1 of [RFC7231], 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 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 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 HTTP header fields. In doing so, uses the VCHAR, SP, DIGIT, ALPHA and DQUOTE rules from [RFC5234]. It also includes the OWS and tchar rules from [RFC7230].

When parsing from 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 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 specify a HTTP header as a structured header, its authors 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 type of the header field itself; either Dictionary (Section 3.2), List (Section 3.1), or Item (Section 3.3).

- Define the semantics of those structures.

- Specify any additional constraints upon the structures used, as well as the consequences when those constraints are violated.

Typically, this means that a header definition will specify the top-level type — Dictionary, List or Item — and then define its allowable types, and constraints upon them. For example, a header defined as a list might have all Integer members, or a mix of types; a header defined as an Item might allow only Strings, and additionally only strings beginning with the letter "Q".

When Structured Headers parsing fails, the header is ignored (see Section 4.2); in most situations, violating header-specific constraints should have the same effect. Thus, if a header is defined as an Item and required to be an Integer, but a String is received, it will by default be ignored. If the header requires different error handling, this should be explicitly specified.

However, both Items and Inner Lists allow parameters as an extensibility mechanism; this means that values can later be extended to accommodate more information, if need be. As a result, header specifications are discouraged from defining the presence of an unrecognised parameter as an error condition.

Conversely, inner lists are only valid when a header definition explicitly allows them.

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, the types allowed in a dictionary’s values, or the number of items in a list). Likewise, header field definitions can only 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.

Specifications can refer to a Structured Header’s field-name as a "structured header name" and its field-value as a "structured header value" as necessary. Header definitions are encouraged to use the ABNF rules beginning with "sh-" defined in this specification; other rules in this specification are not intended for their use.

For example, a fictitious Foo-Example header field might be specified as:
42. Foo-Example Header

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

Foo-Example is a Item Structured Header [RFCxxxx]. Its value MUST be an Integer (Section Y.Y of [RFCxxxx]). Its ABNF is:

   Foo-Example = sh-integer

Its value indicates the amount of Foo in the message, and MUST be between 0 and 10, inclusive; other values MUST cause the entire header to be ignored.

The following parameters are defined:
* A parameter whose name is "fooUrl", and whose value is a string (Section Y.Y of [RFCxxxx]), conveying the Foo URLs for the message. See below for processing requirements.

"fooUrl" contains a URI-reference (Section 4.1 of [RFC3986], Section 4.1). If its value is not a valid URI-reference, that URL MUST be ignored. If its value is a relative reference (Section 4.2 of [RFC3986]), it MUST be resolved (Section 5 of [RFC3986]) before being used.

For example:

   Foo-Example: 2; fooUrl="https://foo.example.com/"

3. Structured 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 headers.

In summary:

* There are three top-level types that a HTTP header can be defined as; Lists, Dictionaries, and Items.

* Lists and Dictionaries are containers; their members can be Items or Inner Lists (which are themselves lists of items).

* Both Items and Inner Lists can be parameterised with key/value pairs.
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 (Section 3.1.1), both of which can be parameterised (Section 3.1.2).

The ABNF for lists in HTTP headers is:

\[
\text{sh-list} = \text{list-member} \ast (\text{OWS} \text{,} \text{OWS} \text{list-member}) \\
\text{list-member} = \text{sh-item} / \text{inner-list}
\]

In 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 HTTP headers, an empty list is denoted by not serialising the header at all.

Parsers MUST support lists containing at least 1024 members. Header specifications can constrain the types and cardinality of individual list values as they require.

3.1.1. Inner Lists

An inner list is an array of zero or more items (Section 3.3). Both the individual items and the inner-list itself can be parameterised (Section 3.1.2).

The ABNF for inner-lists in HTTP headers is:

\[
\text{inner-list} = "(\text{OWS} [ \text{sh-item} \ast (\text{SP OWS sh-item}) \text{OWS}]")" \\
*parameter
\]

In 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 inner-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.

A header field whose value is defined as a list of inner-lists with parameters at both levels could look like:

Example-ListListParam: ("foo"; a=1; b=2); lvl=5, ("bar", "baz"); lvl=1
Parsers MUST support inner-lists containing at least 256 members. Header specifications can constrain the types and cardinality of individual inner-list members as they require.

3.1.2. Parameters

Parameters are an ordered map of key-values pairs that are associated with an item (Section 3.3) or inner-list (Section 3.1.1). The keys are required to be unique within the scope of a map of parameters, and the values are bare items (i.e., they themselves cannot be parameterised; see Section 3.3).

The ABNF for parameters in HTTP headers is:

```
parameter     = ";" OWS param-name [ "=" param-value ]
param-name    = key
key           = lcalpha *( lcalpha / DIGIT / "_" / "-" / "*" )
lcalpha       = %x61-7A ; a-z
param-value   = bare-item
```

In HTTP headers, parameters are separated from their item or inner-list and each other by semicolons. For example:

Example-ParamListHeader: abc;a=1;b=2; cde_456, (ghi;jk=4 l);q="9";r=w

Parsers MUST support at least 256 parameters on an item or inner-list, and support parameter keys with at least 64 characters. Header specifications can constrain the types and cardinality of individual parameter names and values as they require.

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, both of which can be parameterised (Section 3.1.2). 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 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 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)

A dictionary with a mix of singular and list values, some with parameters:

Example-MixDict: a=(1 2), b=3, c=4;aa=bb, d=(5 6);valid=?1

As with lists, an empty dictionary is represented in HTTP headers by omitting the entire header field.

Typically, a header field specification will define the semantics of dictionaries by specifying the allowed type(s) for 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 is can be a integer (Section 3.3.1), float (Section 3.3.2), string (Section 3.3.3), token (Section 3.3.4), byte sequence (Section 3.3.5), or Boolean (Section 3.3.6). It can have associated parameters (Section 3.1.2).

The ABNF for items in HTTP headers is:

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

For example, a header field that is defined to be an Item that is an integer might look like:

Example-IntItemHeader: 5

or with parameters:

Example-IntItemHeader: 5; foo=bar
3.3.1. 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 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.

3.3.2. Floats

Floats are decimal numbers with an integer and a fractional component. The fractional component has at most six digits of precision. Additionally, like integers, it can have no more than fifteen digits in total, which in some cases further constrains its precision.

The ABNF for floats in HTTP headers is:

```
sh-float    = ["-" ] (1*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

3.3.3. Strings

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

The ABNF for strings in HTTP headers is:
In 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 characters after "\" MUST cause parsing to fail.

Unicode is not directly supported in strings, 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 content, a byte sequence (Section 3.3.5) SHOULD be specified, along with a character encoding (preferably [UTF-8]).

Parsers MUST support strings with at least 1024 characters.

3.3.4. Tokens

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

The ABNF for tokens in HTTP headers is:

```
sh-token = ALPHA *( tchar / ":" / "/" )
```

Parsers MUST support tokens with at least 512 characters.

Note that a Structured Header token allows the characters as the "token" ABNF rule defined in [RFC7230], with the exceptions that the first character is required to be ALPHA, and ":" and "/" are also allowed.

3.3.5. Byte Sequences

Byte sequences can be conveyed in Structured Headers.

The ABNF for a byte sequence in HTTP headers is:
sh-binary = "*" *(base64) "*"
base64    = ALPHA / DIGIT / "+" / "/" / ";="

In 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.

3.3.6. Booleans

Boolean values can be conveyed in Structured Headers.

The ABNF for a Boolean in HTTP headers is:

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

In HTTP headers, a boolean is indicated with a leading "?" character followed by a "1" for a true value or "0" for false. For example:

Example-BoolHdr: ?1

4. Working With Structured Headers in HTTP Headers

This section defines how to serialize and parse Structured Headers in 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, return an ASCII string suitable for use in a HTTP header value.

1. If the structure is a dictionary or list and its value is empty (i.e., it has no members), do not serialize the 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 running Serializing a Dictionary (Section 4.1.2) with the structure.

3. Else if the structure is a list, let output_string be the result of running Serializing a List (Section 4.1.1) with the structure.
4. Else if the structure is an item, let output_string be the result of running Serializing an Item (Section 4.1.3) with the structure.

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 an array of (member-value, parameters) tuples as input_list, return an ASCII string suitable for use in a HTTP header value.

1. Let output be an empty string.

2. For each (member-value, parameters) of input_list:
   1. If member-value is an array, append the result of running Serialising an Inner List (Section 4.1.1.1) with (member-value, parameters) to output.
   2. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member-value, parameters) to output.

3. If more member-values remain in input_list:
   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 of (member-value, parameters) tuples as inner_list, and parameters as list_parameters, return an ASCII string suitable for use in a HTTP header value.

1. Let output be the string ":".

2. For each (member-value, parameters) of inner_list:
   1. Append the result of running Serializing an Item (Section 4.1.3) with (member-value, parameters) to output.
   2. If more values remain in inner_list, append a single WS to output.
3. Append ")" to output.

4. Append the result of running Serializing Parameters `Section 4.1.1.2` with list_parameters to output.

5. Return output.

4.1.1.2. Serializing Parameters

Given an ordered dictionary as input_parameters (each member having a param-name and a param-value), return an ASCII string suitable for use in a HTTP header value.

1. Let output be an empty string.

2. For each parameter-name with a value of param-value in input_parameters:
   1. Append ";" to output.
   2. Append the result of running Serializing a Key `Section 4.1.1.3` with param-name to output.

3. If param-value is not null:
   1. Append "=" to output.
   2. Append the result of running Serializing a bare Item `Section 4.1.3.1` with param-value to output.

3. Return output.

4.1.1.3. Serializing a Key

Given a key as input_key, return an ASCII string suitable for use in a HTTP header value.

1. If input_key is not a sequence of characters, or contains characters not in localha, DIGIT, ",", ",", or ",", 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 an ordered dictionary as input_dictionary (each member having a member-name and a tuple value of (member-value, parameters)), return an ASCII string suitable for use in a HTTP header value.

1. Let output be an empty string.

2. For each member-name with a value of (member-value, parameters) in input_dictionary:
   1. Append the result of running Serializing a Key (Section 4.1.1.3) with member’s member-name to output.
   2. Append "=" to output.
   3. If member-value is an array, append the result of running Serialising an Inner List (Section 4.1.1.1) with (member-value, parameters) to output.
   4. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member-value, parameters) to output.

5. 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 bare_item and parameters item_parameters as input, return an ASCII string suitable for use in a HTTP header value.

1. Let output be an empty string.

2. Append the result of running Serializing a Bare Item (Section 4.1.3.1) with bare_item to output.

3. Append the result of running Serializing Parameters (Section 4.1.1.2) with item_parameters to output.

4. Return output.
4.1.3.1. Serialising a Bare Item

Given an item as input_item, return an ASCII string suitable for use in a HTTP header value.

1. If input_item is an integer, return the result of running Serializing an Integer (Section 4.1.4) with input_item.

2. If input_item is a float, return the result of running Serializing a Float (Section 4.1.5) with input_item.

3. If input_item is a string, return the result of running Serializing a String (Section 4.1.6) with input_item.

4. If input_item is a token, return the result of running Serializing a Token (Section 4.1.7) with input_item.

5. If input_item is a Boolean, return the result of running Serializing a Boolean (Section 4.1.9) with input_item.

6. If input_item is a byte sequence, return the result of running Serializing a Byte Sequence (Section 4.1.8) with input_item.

7. Otherwise, fail serialisation.

4.1.4. Serializing an Integer

Given an integer as input_integer, return an ASCII string suitable for use in a HTTP header value.

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, return an ASCII string suitable for use in a HTTP header value.

1. Let output be an empty string.
2. If input_float is less than (but not equal to) 0, append "-" to output.
3. Append input_float’s integer component represented in base 10 (using only decimal digits) to output; if it is zero, append "0".
4. Let integer_digits be the number of characters appended in the previous step.
5. If integer_digits is greater than 14, fail serialisation.
7. Let fractional_digits_avail be the minimum of digits_avail and 6.
8. Append "." to output.
9. Append at most fractional_digits_avail digits of input_float’s fractional component represented in base 10 to output (using only decimal digits, and truncating any remaining digits); if it is zero, append "0".
10. Return output.

4.1.6. Serializing a String

Given a string as input_string, return an ASCII string suitable for use in a HTTP header value.

1. If input_string is not a sequence of characters, or contains characters in the range %x00-1f or %x7f (i.e., is not in 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, return an ASCII string suitable for use in a HTTP header value.

1. If input_token is not a sequence of characters, or contains characters not allowed by the tchar ABNF rule, 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, return an ASCII string suitable for use in a HTTP header value.

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`, return an ASCII string suitable for use in a HTTP header value.

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 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 empty 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 running Parsing a List (Section 4.2.1) with `input_string`.
4. If `header_type` is "dictionary", let output be the result of running Parsing a Dictionary (Section 4.2.2) with `input_string`.
5. If `header_type` is "item", let output be the result of running Parsing an Item (Section 4.2.3) with `input_string`. 
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 ignored (i.e., treated as if the header field were not present in the message). This is intentionally strict, to improve interoperability and safety, and specifications referencing this document are not allowed to loosen this requirement.

Note that this requirement does not apply to an implementation that is not parsing the header field; for example, an intermediary is not required to strip a failing header field from a message before forwarding it.

4.2.1. Parsing a List

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

1. Let members be an empty array.

2. While input_string is not empty:
1. Append the result of running Parsing an Item or Inner List (Section 4.2.1.1) with input_string to members.

2. Discard any leading OWS from input_string.

3. If input_string is empty, return members.

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

5. Discard any leading OWS from input_string.

6. 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 an Item or Inner List

Given an ASCII string as input_string, return the tuple (item_or_inner_list, parameters), where item_or_inner_list can be either a single bare item, or an array of (bare_item, parameters) tuples. input_string is modified to remove the parsed value.

1. If the first character of input_string is ",", return the result of running Parsing an Inner List (Section 4.2.1.2) with input_string.

2. Return the result of running Parsing an Item (Section 4.2.3) with input_string.

4.2.1.2. Parsing an Inner List

Given an ASCII string as input_string, return the tuple (inner_list, parameters), where inner_list is an array of (bare_item, parameters) tuples. 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. Let parameters be the result of running Parsing Parameters (Section 4.2.3.2) with input_string.

3. Return the tuple (inner_list, parameters).

4. Let item be the result of running Parsing an Item (Section 4.2.3) with input_string.

5. Append item to inner_list.

4. The end of the inner list was not found; fail parsing.

4.2.2. Parsing a Dictionary

Given an ASCII string as input_string, return an ordered map whose values are (item_or_inner_list, parameters) tuples. 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 (Section 4.2.3.3) 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. Let member be the result of running Parsing an Item or Inner List (Section 4.2.1.1) with input_string.

   5. Add name this_key with value member 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, there is a trailing comma; fail parsing.

3. No structured data has been found; return dictionary (which is empty).

4.2.3. Parsing an Item

Given an ASCII string as input_string, return a (bare_item, parameters) tuple. input_string is modified to remove the parsed value.

1. Let bare_item be the result of running Parsing a Bare Item (Section 4.2.3.1) with input_string.

2. Let parameters be the result of running Parsing Parameters (Section 4.2.3.2) with input_string.

3. Return the tuple (bare_item, parameters).

4.2.3.1. Parsing a Bare Item

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

1. If the first character of input_string is a "-" or a DIGIT, return the result of running Parsing a Number (Section 4.2.4) with input_string.

2. If the first character of input_string is a DQUOTE, return the result of running Parsing a String (Section 4.2.5) with input_string.

3. If the first character of input_string is ",", return the result of running Parsing a Byte Sequence (Section 4.2.7) with input_string.

4. If the first character of input_string is ",", return the result of running Parsing a Boolean (Section 4.2.8) with input_string.

5. If the first character of input_string is an ALPHA, return the result of running Parsing a Token (Section 4.2.6) with input_string.

6. Otherwise, the item type is unrecognized; fail parsing.
4.2.3.2. Parsing Parameters

Given an ASCII string as input_string, return an ordered map whose values are bare items. input_string is modified to remove the parsed value.

1. Let parameters be an empty, ordered map.
2. While input_string is not empty:
   1. If the first character of input_string is not ";", exit the loop.
   2. Consume a ";" character from the beginning of input_string.
   3. Discard any leading OWS from input_string.
   4. let param_name be the result of running Parsing a Key (Section 4.2.3.3) with input_string.
   5. If param_name is already present in parameters, there is a duplicate; fail parsing.
   6. Let param_value be a null value.
   7. 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 running Parsing a Bare Item (Section 4.2.3.1) with input_string.
   8. Append key param_name with value param_value to parameters.
3. Return parameters.

4.2.3.3. Parsing a Key from Text

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

1. If the first character of input_string is not localpha, fail parsing.
2. Let output_string be an empty string.
3. While input_string is not empty:
1. If the first character of input_string is not one of lcalpha, DIGIT, "+", ",", or "-", return output_string.
2. Let char be the result of removing the first character of input_string.
3. Append char to output_string.
4. Return output_string.

4.2.4. Parsing a Number from Text

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

NOTE: This algorithm parses both Integers (Section 3.3.1) and Floats (Section 3.3.2), 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 3.3.1, fail parsing.

9. Otherwise:
   1. If the final character of input_number is ".", fail parsing.
   2. If the number of characters after "." in input_number is greater than six, fail parsing.
   3. Parse input_number as a float and let output_number be the product of the result and sign.

10. Return output_number.

4.2.5. Parsing a String from Text

Given an ASCII string as 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. Let next_char be the result of consuming the first character of input_string.
      3. If next_char is not DQUOTE or "\", fail parsing.
4. 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.

4.2.6. Parsing a Token from Text

Given an ASCII string as 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. If the first character of input_string is not allowed by the tchar ABNF rule, return output_string.

   2. Let char be the result of consuming the first character of input_string.

   3. Append char to output_string.

4. Return output_string.

4.2.7. Parsing a Byte Sequence from Text

Given an ASCII string as 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.

4.2.8. Parsing a Boolean from Text

   Given an ASCII string as 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


7.2. Informative References


7.3. URIs

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


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

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-13

- Editorial improvements.
- Define "structured header name" and "structured header value" terms (#908).
- Corrected text about valid characters in strings (#931).
- Removed most instances of the word "textual", as it was redundant (#915).
- Allowed parameters on Items and Inner Lists (#907).
- Expand the range of characters in token (#961).

D.2. Since draft-ietf-httpbis-header-structure-12

- Editorial improvements.
- Reworked float serialisation (#896).
- Don’t add a trailing space in inner-list (#904).


- Allow * in key (#844).
- Constrain floats to six digits of precision (#848).
- Allow dictionary members to have parameters (#842).

D.4. Since draft-ietf-httpbis-header-structure-10

- Update abstract (#799).
- Input and output are now arrays of bytes (#662).
- Implementations need to preserve difference between token and string (#790).
- Allow empty dictionaries and lists (#781).
- 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.5. 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.6. 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.7. 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.8. 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.9. 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.10. Since draft-ietf-httpbis-header-structure-04

o Remove identifiers from item.

o Remove most limits on sizes.

o Refine number parsing.

D.11. Since draft-ietf-httpbis-header-structure-03

o Strengthen language around failure handling.

D.12. 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.
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.

---

D.13. Since draft-ietf-httpbis-header-structure-01

- Replaced with draft-nottingham-structured-headers.

D.14. 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