Stream: Internet Engineering Task Force (IETF)
RFC: 8941
Category: Standards Track
Published: October 2020
ISSN: 2070-1721
Authors: M. Nottingham, *Fastly*; P-H. Kamp, *The Varnish Cache Project*

RFC 8941
Structured Field Values for HTTP

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 and trailer fields, known as "Structured Fields", "Structured Headers", or "Structured Trailers". It is intended for use by specifications of new HTTP fields that wish to use a common syntax that is more restrictive than traditional HTTP field values.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8941.

Table of Contents

1. Introduction
   1.1. Intentionally Strict Processing
   1.2. Notational Conventions
2. Defining New Structured Fields
3. Structured Data Types
   3.1. Lists
      3.1.1. Inner Lists
      3.1.2. Parameters
   3.2. Dictionaries
   3.3. Items
      3.3.1. Integers
      3.3.2. Decimals
      3.3.3. Strings
      3.3.4. Tokens
      3.3.5. Byte Sequences
      3.3.6. Booleans
4. Working with Structured Fields in HTTP
   4.1. Serializing Structured Fields
      4.1.1. Serializing a List
      4.1.2. Serializing a Dictionary
      4.1.3. Serializing an Item
      4.1.4. Serializing an Integer
      4.1.5. Serializing a Decimal
      4.1.6. Serializing a String
      4.1.7. Serializing a Token
      4.1.8. Serializing a Byte Sequence
      4.1.9. Serializing a Boolean

4.2. Parsing Structured Fields
4.2.1. Parsing a List
4.2.2. Parsing a Dictionary
4.2.3. Parsing an Item
4.2.4. Parsing an Integer or Decimal
4.2.5. Parsing a String
4.2.6. Parsing a Token
4.2.7. Parsing a Byte Sequence
4.2.8. Parsing a Boolean
5. IANA Considerations
6. Security Considerations
7. References
7.1. Normative References
7.2. Informative References
Appendix A. Frequently Asked Questions
A.1. Why Not JSON?
Appendix B. Implementation Notes
Acknowledgements
Authors' Addresses

1. Introduction

Specifying the syntax of new HTTP header (and trailer) 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 field author.

Once a field is defined, bespoke parsers and serializers often need to be written, because each field value has a 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 field values to address these problems. In particular, it defines a generic, abstract model for them, along with a concrete serialization for expressing that model in HTTP [RFC7230] header and trailer fields.

An HTTP field that is defined as a "Structured Header" or "Structured Trailer" (if the field can be either, it is a "Structured Field") uses the types defined in this specification to define its syntax and basic handling rules, thereby simplifying both its definition by specification writers and handling by implementations.

Additionally, future versions of HTTP can define alternative serializations of the abstract model of these structures, allowing fields that use that model 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 fields; the mechanisms described herein are only intended to be used with fields that explicitly opt into them.

Section 2 describes how to specify a Structured Field.

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

Those abstract types can be serialized into and parsed from HTTP field values using the algorithms described in Section 4.

1.1. Intentionally Strict Processing

This specification intentionally defines strict parsing and serialization behaviors 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 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 field value 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 serialization behaviors and the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] to illustrate expected syntax in HTTP header fields. In doing so, it uses the VCHAR, SP, DIGIT, ALPHA, and DQUOTE rules from [RFC5234]. It also includes the tchar and OWS rules from [RFC7230].

When parsing from HTTP fields, implementations **MUST** have behavior that is indistinguishable from following the algorithms. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence.

For serialization to HTTP fields, the ABNF illustrates their expected wire representations, and the algorithms define the recommended way to produce them. Implementations **MAY** vary from the specified behavior so long as the output is still correctly handled by the parsing algorithm.

2. Defining New Structured Fields

To specify an HTTP field as a Structured Field, its authors need to:

- Normatively reference this specification. Recipients and generators of the field need to know that the requirements of this document are in effect.
- Identify whether the field is a Structured Header (i.e., it can only be used in the header section -- the common case), a Structured Trailer (only in the trailer section), or a Structured Field (both).
- Specify the type of the field value; either List (Section 3.1), Dictionary (Section 3.2), or Item (Section 3.3).
- Define the semantics of the field value.
- Specify any additional constraints upon the field value, as well as the consequences when those constraints are violated.

Typically, this means that a field definition will specify the top-level type -- List, Dictionary, 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", or strings in lowercase. Likewise, Inner Lists (Section 3.1.1) are only valid when a field definition explicitly allows them.

When parsing fails, the entire field is ignored (see Section 4.2); in most situations, violating field-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, the field will by default be ignored. If the field requires different error handling, this should be explicitly specified.

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. To preserve forward compatibility, field specifications are discouraged from defining the presence of an unrecognized parameter as an error condition.

To further assure that this extensibility is available in the future, and to encourage consumers to use a complete parser implementation, a field definition can specify that "grease" parameters be added by senders. A specification could stipulate that all parameters that fit a defined pattern are reserved for this use and then encourage them to be sent on some portion of requests. This helps to discourage recipients from writing a parser that does not account for Parameters.

Specifications that use Dictionaries can also allow for forward compatibility by requiring that the presence of -- as well as value and type associated with -- unknown members be ignored. Subsequent specifications can then add additional members, specifying constraints on them as appropriate.

An extension to a Structured Field can then require that an entire field value be ignored by a recipient that understands the extension if constraints on the value it defines are not met.

A 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 Decimals, the format of Strings and Tokens, the types allowed in a Dictionary's values, or the number of Items in a List). Likewise, field definitions can only use this specification for the entire field value, not a portion thereof.

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

Specifications can refer to a field name as a "structured header name", "structured trailer name", or "structured field name" as appropriate. Likewise, they can refer its field value as a "structured header value", "structured trailer value", or "structured field value" as necessary. Field definitions are encouraged to use the ABNF rules beginning with "sf-" defined in this specification; other rules in this specification are not intended to be used in field definitions.

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 an Item Structured Header [RFCxxxx]. Its value MUST be an Integer (Section Y.Y of [RFCxxxx]). Its ABNF is:
>
> ```
> Foo-Example = sf-integer
> ```
>
> Its value indicates the amount of Foo in the message, and it MUST be between 0 and 10, inclusive; other values MUST cause the entire header field to be ignored.
>
> The following parameter is defined:
>
> - A parameter whose name is "foourl", and whose value is a String (Section Y.Y of [RFCxxxx]), conveying the Foo URL for the message. See below for processing requirements.

"foourl" contains a URI-reference (Section 4.1 of [RFC3986]). If its value is not a valid URI-reference, the entire header field 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 types for Structured Fields. The ABNF provided represents the on-wire format in HTTP field values.

In summary:

- There are three top-level types that an HTTP field can be defined as: Lists, Dictionaries, and Items.
- Lists and Dictionaries are containers; their members can be Items or Inner Lists (which are themselves arrays of Items).
- Both Items and Inner Lists can be Parameterized 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 Parameterized (Section 3.1.2).

The ABNF for Lists in HTTP fields is:

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

Each member is separated by a comma and optional whitespace. For example, a field whose value is defined as a List of Strings could look like:

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

An empty List is denoted by not serializing the field at all. This implies that fields defined as Lists have a default empty value.

Note that Lists can have their members split across multiple lines inside a header or trailer section, as per Section 3.2.2 of [RFC7230]; for example, the following are equivalent:

```
Example-Hdr: foo, bar
```

and

```
Example-Hdr: foo
Example-Hdr: bar
```

However, individual members of a List cannot be safely split between lines; see Section 4.2 for details.

Parsers **MUST** support Lists containing at least 1024 members. Field 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 Parameterized (Section 3.1.2).

The ABNF for Inner Lists is:

```
inner-list    = "(" *SP [ sf-item *( 1*SP sf-item ) *SP ] ")"
                parameters
```

Inner Lists are denoted by surrounding parenthesis, and their values are delimited by one or more spaces. A field whose value is defined as a List of Inner Lists of Strings could look like:

```
Example-StrListList: ("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. Field 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-value pairs that are associated with an Item (Section 3.3) or Inner List (Section 3.1.1). The keys are unique within the scope of the Parameters they occur within, and the values are bare items (i.e., they themselves cannot be parameterized; see Section 3.3).

The ABNF for Parameters is:

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

Note that parameters are ordered as serialized, and parameter keys cannot contain uppercase letters. A parameter is separated from its Item or Inner List and other parameters by a semicolon. For example:

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

Parameters whose value is Boolean (see Section 3.3.6) true **MUST** omit that value when serialized. For example, the "a" parameter here is true, while the "b" parameter is false:

```
Example-Int: 1; a; b=?0
```

Note that this requirement is only on serialization; parsers are still required to correctly handle the true value when it appears in a parameter.

Parsers **MUST** support at least 256 parameters on an Item or Inner List, and support parameter keys with at least 64 characters. Field specifications can constrain the order of individual parameters, as well as their values' types as required.

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 Parameterized (Section 3.1.2). There can be zero or more members, and their names are unique in 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 is:

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

Members are ordered as serialized and separated by a comma with optional whitespace. Member names cannot contain uppercase characters. Names and values are separated by "=" (without whitespace). For example:

```
Example-Dict: en="Applepie", da=:w4ZibGV0w6ZydGU=:
```

Note that in this example, the final "=" is due to the inclusion of a Byte Sequence; see Section 3.3.5.

Members whose value is Boolean (see Section 3.3.6) true **MUST** omit that value when serialized. For example, here both "b" and "c" are true:

```
Example-Dict: a=?0, b, c; foo=bar
```

Note that this requirement is only on serialization; parsers are still required to correctly handle the true Boolean value when it appears in Dictionary values.

A Dictionary with a member whose value is an Inner List of Tokens:

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

A Dictionary with a mix of Items and Inner Lists, some with parameters:

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

As with Lists, an empty Dictionary is represented by omitting the entire field. This implies that fields defined as Dictionaries have a default empty value.

Typically, a field specification will define the semantics of Dictionaries by specifying the allowed type(s) for individual members by their names, as well as whether their presence is required or optional. Recipients **MUST** ignore names that are undefined or unknown, unless the field's specification specifically disallows them.

Note that Dictionaries can have their members split across multiple lines inside a header or trailer section; for example, the following are equivalent:

```
Example-Hdr: foo=1, bar=2
```

and

```
Example-Hdr: foo=1
Example-Hdr: bar=2
```

However, individual members of a Dictionary cannot be safely split between lines; see Section 4.2 for details.

Parsers **MUST** support Dictionaries containing at least 1024 name/value pairs and names with at least 64 characters. Field specifications can constrain the order of individual Dictionary members, as well as their values' types as required.

3.3. Items

An Item can be an Integer (Section 3.3.1), a Decimal (Section 3.3.2), a String (Section 3.3.3), a Token (Section 3.3.4), a Byte Sequence (Section 3.3.5), or a Boolean (Section 3.3.6). It can have associated parameters (Section 3.1.2).

The ABNF for Items is:

```
sf-item   = bare-item parameters
bare-item = sf-integer / sf-decimal / sf-string / sf-token
            / sf-binary / sf-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-IntItem: 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 is:

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

For example:

```
Example-Integer: 42
```

Integers larger than 15 digits can be supported in a variety of ways; for example, by using a String (Section 3.3.3), a Byte Sequence (Section 3.3.5), or a parameter on an Integer that acts as a scaling factor.

While it is possible to serialize Integers with leading zeros (e.g., "0002", "-01") and signed zero ("-0"), these distinctions may not be preserved by implementations.

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. Decimals

Decimals are numbers with an integer and a fractional component. The integer component has at most 12 digits; the fractional component has at most three digits.

The ABNF for decimals is:

```
sf-decimal  = ["-"] 1*12DIGIT "." 1*3DIGIT
```

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

```
Example-Decimal: 4.5
```

While it is possible to serialize Decimals with leading zeros (e.g., "0002.5", "-01.334"), trailing zeros (e.g., "5.230", "-0.40"), and signed zero (e.g., "-0.0"), these distinctions may not be preserved by implementations.

Note that the serialization algorithm (Section 4.1.5) rounds input with more than three digits of precision in the fractional component. If an alternative rounding strategy is desired, this should be specified by the header definition to occur before serialization.

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 is:

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

Strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

```
Example-String: "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 -- field 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) can be specified, along with a character encoding (preferably [STD63]).

Parsers **MUST** support Strings (after any decoding) 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 field value serialization.

The ABNF for Tokens is:

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

For example:

```
Example-Token: foo123/456
```

Parsers **MUST** support Tokens with at least 512 characters.

Note that Token allows the same characters as the "token" ABNF rule defined in [RFC7230], with the exceptions that the first character is required to be either ALPHA or "*", and ":" and "/" are also allowed in subsequent characters.

3.3.5. Byte Sequences

Byte Sequences can be conveyed in Structured Fields.

The ABNF for a Byte Sequence is:

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

A Byte Sequence is delimited with colons and encoded using base64 ([RFC4648], Section 4). For example:

```
Example-Binary: :cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==:
```

Parsers **MUST** support Byte Sequences with at least 16384 octets after decoding.

3.3.6. Booleans

Boolean values can be conveyed in Structured Fields.

The ABNF for a Boolean is:

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

A Boolean is indicated with a leading "?" character followed by a "1" for a true value or "0" for false. For example:

```
Example-Bool: ?1
```

Note that in Dictionary (Section 3.2) and Parameter (Section 3.1.2) values, Boolean true is indicated by omitting the value.

4. Working with Structured Fields in HTTP

This section defines how to serialize and parse Structured Fields in textual HTTP field values and other encodings compatible with them (e.g., in HTTP/2 [RFC7540] before compression with HPACK [RFC7541]).

4.1. Serializing Structured Fields

Given a structure defined in this specification, return an ASCII string suitable for use in an HTTP field 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 List, let output_string be the result of running Serializing a List (Section 4.1.1) with the structure.

3. Else, if the structure is a Dictionary, let output_string be the result of running Serializing a Dictionary (Section 4.1.2) 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 serialization.
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 an HTTP field 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 Serializing 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 "," to output.
        2. Append a single SP to output.
3. Return output.

4.1.1.1. Serializing 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 an HTTP field 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 SP 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 an HTTP field value.

1. Let output be an empty string.
2. For each param_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 Boolean true:
        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 an HTTP field value.

1. Convert input_key into a sequence of ASCII characters; if conversion fails, fail serialization.
2. If input_key contains characters not in lcalpha, DIGIT, "_", "-", ".", or "*", fail serialization.
3. If the first character of input_key is not lcalpha or "*", fail serialization.
4. Let output be an empty string.
5. Append input_key to output.
6. 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 an HTTP field 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. If member_value is Boolean true:
        1. Append the result of running Serializing Parameters (Section 4.1.1.2) with parameters to output.

3. Otherwise:
    1. Append "=" to output.
    2. If member_value is an array, append the result of running Serializing an Inner List (Section 4.1.1.1) with (member_value, parameters) to output.
    3. Otherwise, append the result of running Serializing an Item (Section 4.1.3) with (member_value, parameters) to output.
4. If more members remain in input_dictionary:
    1. Append "," to output.
    2. Append a single SP to output.
3. Return output.

4.1.3. Serializing an Item

Given an Item as bare_item and Parameters as item_parameters, return an ASCII string suitable for use in an HTTP field 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. Serializing a Bare Item

Given an Item as input_item, return an ASCII string suitable for use in an HTTP field 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 Decimal, return the result of running Serializing a Decimal (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 Byte Sequence, return the result of running Serializing a Byte Sequence (Section 4.1.8) with input_item.
6. If input_item is a Boolean, return the result of running Serializing a Boolean (Section 4.1.9) with input_item.
7. Otherwise, fail serialization.

4.1.4. Serializing an Integer

Given an Integer as input_integer, return an ASCII string suitable for use in an HTTP field 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 serialization.
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 Decimal

Given a decimal number as input_decimal, return an ASCII string suitable for use in an HTTP field value.

1. If input_decimal is not a decimal number, fail serialization.
2. If input_decimal has more than three significant digits to the right of the decimal point, round it to three decimal places, rounding the final digit to the nearest value, or to the even value if it is equidistant.
3. If input_decimal has more than 12 significant digits to the left of the decimal point after rounding, fail serialization.
4. Let output be an empty string.
5. If input_decimal is less than (but not equal to) 0, append "-" to output.
6. Append input_decimal's integer component represented in base 10 (using only decimal digits) to output; if it is zero, append "0".
7. Append "." to output.
8. If input_decimal's fractional component is zero, append "0" to output.
9. Otherwise, append the significant digits of input_decimal's fractional component represented in base 10 (using only decimal digits) to output.
10. Return output.

4.1.6. Serializing a String

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

1. Convert input_string into a sequence of ASCII characters; if conversion fails, fail serialization.
2. If input_string contains characters in the range %x00-1f or %x7f (i.e., not in VCHAR or SP), fail serialization.
3. Let output be the string DQUOTE.

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 an HTTP field value.

1. Convert input_token into a sequence of ASCII characters; if conversion fails, fail serialization.
2. If the first character of input_token is not ALPHA or "*", or the remaining portion contains a character not in tchar, ":", or "/", fail serialization.
3. Let output be an empty string.
4. Append input_token to output.
5. Return output.

4.1.8. Serializing a Byte Sequence

Given a Byte Sequence as input_bytes, return an ASCII string suitable for use in an HTTP field value.

1. If input_bytes is not a sequence of bytes, fail serialization.
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 an HTTP field value.

1. If input_boolean is not a boolean, fail serialization.
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 Structured Fields

When a receiving implementation parses HTTP fields that are known to be Structured Fields, 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 as input_bytes that represent the chosen field's field-value (which is empty if that field is not present) and field_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 SP characters from input_string.
3. If field_type is "list", let output be the result of running Parsing a List (Section 4.2.1) with input_string.
4. If field_type is "dictionary", let output be the result of running Parsing a Dictionary (Section 4.2.2) with input_string.
5. If field_type is "item", let output be the result of running Parsing an Item (Section 4.2.3) with input_string.
6. Discard any leading SP characters from input_string.
7. If input_string is not empty, fail parsing.
8. Otherwise, return output.

When generating input_bytes, parsers **MUST** combine all field lines in the same section (header or trailer) that case-insensitively match the field name into one comma-separated field-value, as per [RFC7230], Section 3.2.2; this assures that the entire field value is processed correctly.

For Lists and Dictionaries, this has the effect of correctly concatenating all of the field's lines, as long as individual members of the top-level data structure are not split across multiple header instances. The parsing algorithms for both types allow tab characters, since these might be used to combine field lines by some implementations.

Strings split across multiple field lines will have unpredictable results, because one or more commas (with optional whitespace) 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, even when they are both under the control of the same party.

Tokens, Integers, Decimals, and Byte Sequences cannot be split across multiple field lines because the inserted commas will cause parsing to fail.

Parsers **MAY** fail when processing a field value spread across multiple field lines, when one of those lines does not parse as that field. For example, a parsing handling an Example-String field that's defined as an sf-string is allowed to fail when processing this field section:

```
Example-String: "foo
Example-String: bar"
```

If parsing fails -- including when calling another algorithm -- the entire field value **MUST** be ignored (i.e., treated as if the field were not present in the section). 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 field; for example, an intermediary is not required to strip a failing 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 characters from input_string.
    3. If input_string is empty, return members.
    4. Consume the first character of input_string; if it is not ",", fail parsing.
    5. Discard any leading OWS characters 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 SP characters 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).
    3. Let item be the result of running Parsing an Item (Section 4.2.3) 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

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 the first character of input_string is "=":
        1. Consume the first character of input_string.
        2. Let member be the result of running Parsing an Item or Inner List (Section 4.2.1.1) with input_string.
    3. Otherwise:
        1. Let value be Boolean true.
        2. Let parameters be the result of running Parsing Parameters (Section 4.2.3.2) with input_string.
        3. Let member be the tuple (value, parameters).

4. Add name this_key with value member to dictionary. If dictionary already contains a name this_key (comparing character for character), overwrite its value.
5. Discard any leading OWS characters from input_string.
6. If input_string is empty, return dictionary.
7. Consume the first character of input_string; if it is not ",", fail parsing.
8. Discard any leading OWS characters from input_string.
9. If input_string is empty, there is a trailing comma; fail parsing.

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

Note that when duplicate Dictionary keys are encountered, all but the last instance are ignored.

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 an Integer or Decimal (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 an ALPHA or "*", return the result of running Parsing a Token (Section 4.2.6) with input_string.
4. If the first character of input_string is ":", return the result of running Parsing a Byte Sequence (Section 4.2.7) with input_string.
5. If the first character of input_string is "?", return the result of running Parsing a Boolean (Section 4.2.8) 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 SP characters from input_string.
    4. Let param_name be the result of running Parsing a Key (Section 4.2.3.3) with input_string.
    5. Let param_value be Boolean true.
    6. 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.
    7. Append key param_name with value param_value to parameters. If parameters already contains a name param_name (comparing character for character), overwrite its value.
3. Return parameters.

Note that when duplicate parameter keys are encountered, all but the last instance are ignored.

4.2.3.3. Parsing a Key

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 lcalpha or "*", 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 consuming the first character of input_string.
    3. Append char to output_string.
4. Return output_string.

4.2.4. Parsing an Integer or Decimal

Given an ASCII string as input_string, return an Integer or Decimal. input_string is modified to remove the parsed value.

NOTE: This algorithm parses both Integers (Section 3.3.1) and Decimals (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 ".":
        1. If input_number contains more than 12 characters, fail parsing.
        2. Otherwise, append char to input_number and set type to "decimal".
    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 "decimal" 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 -999,999,999,999,999 to 999,999,999,999,999 inclusive, 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 three, fail parsing.
    3. Parse input_number as a decimal number and let output_number be the product of the result and sign.
10. Return output_number.

4.2.5. Parsing a String

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., it 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

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 or "*", 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 in tchar, ":", or "/", 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

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 base64-decoding [RFC4648] b64_content, synthesizing padding if necessary (note the requirements about recipient behavior below).
8. Return binary_content.

Because some implementations of base64 do not allow rejection of encoded data that is not properly "=" padded (see [RFC4648], Section 3.2), parsers **SHOULD NOT** fail when "=" padding 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 non-zero pad bits are present, unless they cannot be configured to do so.

This specification does not relax the requirements in [RFC4648], Sections 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

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 document has no IANA actions.

6. Security Considerations

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

It is possible for parties with the ability to inject new HTTP fields to change the meaning of a Structured Field. 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", DOI 10.1109/IEEESTD.2019.8766229, IEEE 754-2019, July 2019, <https://ieeexplore.ieee.org/document/8766229>.

[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>.

[STD63] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003, <https://www.rfc-editor.org/info/std63>.

Appendix A. Frequently Asked Questions

A.1. Why Not JSON?

Earlier proposals for Structured Fields 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 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/serializer on field values.

Since a major goal for Structured Fields 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 fields.

Appendix B. Implementation Notes

A generic implementation of this specification should expose the top-level serialize (Section 4.1) and parse (Section 4.2) 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-field-tests>.

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

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.

The serialization algorithm is defined in a way that it is not strictly limited to the data types defined in Section 3 in every case. For example, Decimals are designed to take broader input and round to allowed values.

Implementations are allowed to limit the size of different structures, subject to the minimums defined for each type. When a structure exceeds an implementation limit, that structure fails parsing or serialization.

Acknowledgements

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

Thanks also to Ian Clelland, Roy Fielding, Anne van Kesteren, Kazuho Oku, Evert Pot, Julian Reschke, Martin Thomson, Mike West, and Jeffrey Yasskin for their contributions.

Authors' Addresses

Mark Nottingham
Fastly
Prahran VIC
Australia
Email: mnot@mnot.net
URI: https://www.mnot.net/

Poul-Henning Kamp
The Varnish Cache Project
Email: phk@varnish-cache.org