Stream:	Internet Engineering Task Force (IETF)	
RFC:	9651	
Obsoletes:	8941	
Category:	Standards Track	
Published:	September 2024	
ISSN:	2070-1721	
Authors:	M. Nottingham	P-H. Kamp
	Cloudflare	*The Varnish Cache Project*

RFC 9651
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.

This document obsoletes RFC 8941.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction 4
1.1. Intentionally Strict Processing 4
1.2. Notational Conventions 5
2. Defining New Structured Fields 5
2.1. Example 6
2.2. Error Handling 7
2.3. Preserving Extensibility 7
2.4. Using New Structured Types in Extensions 8
3. Structured Data Types 8
3.1. Lists 8
3.1.1. Inner Lists 9
3.1.2. Parameters 10
3.2. Dictionaries 10
3.3. Items 12
3.3.1. Integers 12
3.3.2. Decimals 12
3.3.3. Strings 13
3.3.4. Tokens 13
3.3.5. Byte Sequences 14
3.3.6. Booleans 14
3.3.7. Dates 14
3.3.8. Display Strings 14
4. Working with Structured Fields in HTTP 15
4.1. Serializing Structured Fields 15
4.1.1. Serializing a List 16
4.1.2. Serializing a Dictionary 17

4.1.3. Serializing an Item 18
4.1.4. Serializing an Integer 18
4.1.5. Serializing a Decimal 19
4.1.6. Serializing a String 19
4.1.7. Serializing a Token 19
4.1.8. Serializing a Byte Sequence 20
4.1.9. Serializing a Boolean 20
4.1.10. Serializing a Date 20
4.1.11. Serializing a Display String 21
4.2. Parsing Structured Fields 21
4.2.1. Parsing a List 22
4.2.2. Parsing a Dictionary 23
4.2.3. Parsing an Item 24
4.2.4. Parsing an Integer or Decimal 26
4.2.5. Parsing a String 27
4.2.6. Parsing a Token 27
4.2.7. Parsing a Byte Sequence 27
4.2.8. Parsing a Boolean 28
4.2.9. Parsing a Date 28
4.2.10. Parsing a Display String 29
5. IANA Considerations 29
6. Security Considerations 30
7. References 31
7.1. Normative References 31
7.2. Informative References 31
Appendix A. Frequently Asked Questions 32
A.1. Why Not JSON? 32
Appendix B. Implementation Notes 33
Appendix C. ABNF 33
Appendix D. Changes from RFC 8941 34

Acknowledgements 35
Authors' Addresses 35

1. Introduction

Specifying the syntax of new HTTP header (and trailer) fields is an onerous task; even with the guidance in Section 16.3.2 of [HTTP], 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 [HTTP] 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 entire operation altogether.

It is designed to encourage faithful implementation and 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 the VCHAR, SP, DIGIT, ALPHA, and DQUOTE rules from [RFC5234] to specify characters and/or their corresponding ASCII bytes, depending on context. It uses the tchar and OWS rules from [HTTP] for the same purpose.

This document uses algorithms to specify parsing and serialization behaviors. When parsing from HTTP fields, implementations **MUST** have behavior that is indistinguishable from following the algorithms.

For serialization to HTTP fields, 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 described in Section 4.2.

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.

Fields that use the Display String type are advised to carefully specify their allowable Unicode code points; for example, specifying the use of a profile from [PRECIS].

Field definitions can only use this specification for the entire field value, not a portion thereof.

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.

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.

2.1. Example

A fictitious Foo-Example header field might be specified as:

> 42. Foo-Example Header Field
>
> The Foo-Example HTTP header field conveys information about how much Foo the message has.
>
> Foo-Example is an Item Structured Header Field [RFC9651]. Its value **MUST** be an Integer (Section 3.3.1 of [RFC9651]).
>
> 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 key is "foourl", and whose value is a String (Section 3.3.3 of [RFC9651]), 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/"
```

2.2. Error Handling

When parsing fails, the entire field is ignored (see Section 4.2). Field definitions cannot override this 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).

When field-specific constraints are violated, the entire field is also ignored, unless the field definition defines other handling requirements. For example, if a header field is defined as an Item and required to be an Integer, but a String is received, it should be ignored unless that field's definition explicitly specifies otherwise.

2.3. Preserving Extensibility

Structured Fields are designed to be extensible because experience has shown that, even when it is not foreseen, it is often necessary to modify and add to the allowable syntax and semantics of a field in a controlled fashion.

Both Items and Inner Lists allow Parameters as an extensibility mechanism; this means that their 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.

Field specifications are required to be either an Item, List, or Dictionary to preserve extensibility. Fields that erroneously defined as another type (e.g., Integer) are assumed to be Items (i.e., they allow Parameters).

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 keys be ignored. Subsequent specifications can then add additional keys, 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.

2.4. Using New Structured Types in Extensions

Because a field definition needs to reference a specific RFC for Structured Fields, the types available for use in its value are limited to those defined in that RFC. For example, a field whose definition references this document can have a value that uses the Date type (Section 3.3.7), whereas a field whose definition references RFC 8941 cannot because it will be treated as invalid (and therefore discarded) by implementations of that specification.

This limitation also applies to future extensions to a field; for example, a field that is defined with a reference to RFC 8941 cannot use the Date type because some recipients might still be using a parser based on RFC 8941 to process it.

However, this document is designed to be backward compatible with RFC 8941; a parser that implements the requirements here can also parse valid Structured Fields whose definitions reference RFC 8941.

Upgrading a Structured Fields implementation to support a newer revision of the specification (such as this document) brings the possibility that some field values that were invalid according to the earlier RFC might become valid when processed.

For example, a field instance might contain a syntactically valid Date (Section 3.3.7), even though that field's definition does not accommodate Dates. An implementation based on RFC 8941 would fail parsing such a field instance because it is not defined in that specification. If that implementation were upgraded to this specification, parsing would now succeed. In some cases, the resulting Date value will be rejected by field-specific logic, but values in fields that are otherwise ignored (such as extension parameters) might not be detected, and the field might subsequently be accepted and processed.

3. Structured Data Types

This section provides an overview of the abstract types that Structured Fields use and gives a brief description and examples of how each of those types are serialized into textual HTTP fields. Section 4 specifies the details of how they are parsed from and serialized into textual HTTP fields.

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).

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

When serialized as a textual HTTP field, each member is separated by a comma and optional whitespace. For example, a field whose value is defined as a List of Tokens could look like:

```
Example-List: sugar, tea, rum
```

Note that Lists can have their members split across multiple lines of the same header or trailer section, as per Section 5.3 of [HTTP]; for example, the following are equivalent:

```
Example-List: sugar, tea, rum
```

and

```
Example-List: sugar, tea
Example-List: rum
```

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).

When serialized in a textual HTTP field, 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-List: ("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-List: ("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).

Implementations **MUST** provide access to Parameters both by index and by key. Specifications **MAY** use either means of accessing them.

Note that parameters are ordered, and parameter keys cannot contain uppercase letters.

When serialized in a textual HTTP field, a Parameter is separated from its Item or Inner List and other Parameters by a semicolon. For example:

```
Example-List: 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-Integer: 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 key-value pairs, where the keys 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 keys are unique in the scope of the Dictionary they occur within.

Implementations **MUST** provide access to Dictionaries both by index and by key. Specifications **MAY** use either means of accessing the members.

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 keys, as well as whether their presence is required or optional. Recipients **MUST** ignore members whose keys are undefined or unknown, unless the field's specification specifically disallows them.

When serialized as a textual HTTP field, members are ordered as serialized and separated by a comma with optional whitespace. Member keys cannot contain uppercase characters. Keys 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-Dict: rating=1.5, feelings=(joy sadness)
```

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

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

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

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

and

```
Example-Dict: foo=1
Example-Dict: 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 key/value pairs and keys 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), a Boolean (Section 3.3.6), or a Date (Section 3.3.7). It can have associated parameters (Section 3.1.2).

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

```
Example-Integer: 5
```

or with parameters:

```
Example-Integer: 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].

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.

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 field 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.

Non-ASCII characters are not directly supported in Strings because they cause a number of interoperability issues, and -- with few exceptions -- field values do not require them.

When it is necessary for a field value to convey non-ASCII content, a Display String (Section 3.3.8) can be specified.

When serialized in a textual HTTP field, 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.

Parsers **MUST** support Strings (after any decoding) with at least 1024 characters.

3.3.4. Tokens

Tokens are short textual words that begin with an alphabetic character or "*", followed by zero to many token characters, which are the same as those allowed by the "token" ABNF rule defined in [HTTP] plus the ":" and "/" characters.

For example:

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

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

Note that Tokens are defined largely for compatibility with the data model of existing HTTP fields and may require additional steps to use in some implementations. As a result, new fields are encouraged to use Strings.

3.3.5. Byte Sequences

Byte Sequences can be conveyed in Structured Fields.

When serialized in a textual HTTP field, a Byte Sequence is delimited with colons and encoded using base64 ([RFC4648], Section 4). For example:

```
Example-ByteSequence: :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.

When serialized in a textual HTTP field, a Boolean is indicated with a leading "?" character followed by a "1" for a true value or "0" for false. For example:

```
Example-Boolean: ?1
```

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

3.3.7. Dates

Date values can be conveyed in Structured Fields.

Dates have a data model that is similar to Integers, representing a (possibly negative) delta in seconds from 1970-01-01T00:00:00Z, excluding leap seconds. Accordingly, their serialization in textual HTTP fields is similar to that of Integers, distinguished from them with a leading "@".

For example:

```
Example-Date: @1659578233
```

Parsers **MUST** support Dates whose values include all days in years 1 to 9999 (i.e., -62,135,596,800 to 253,402,214,400 delta seconds from 1970-01-01T00:00:00Z).

3.3.8. Display Strings

Display Strings are similar to Strings, in that they consist of zero or more characters, but they allow Unicode scalar values (i.e., all Unicode code points except for surrogates), unlike Strings.

Display Strings are intended for use in cases where a value is displayed to end users and therefore may need to carry non-ASCII content. It is **NOT RECOMMENDED** that they be used in situations where a String (Section 3.3.3) or Token (Section 3.3.4) would be adequate because Unicode has processing considerations (e.g., normalization) and security considerations (e.g., homograph attacks) that make it more difficult to handle correctly.

Note that Display Strings do not indicate the language used in the value; that can be done separately if necessary (e.g., with a parameter).

In textual HTTP fields, Display Strings are represented in a manner similar to Strings, except that non-ASCII characters are percent-encoded; there is a leading "%" to distinguish them from Strings.

For example:

```
Example-DisplayString: %"This is intended for display to %c3%bcsers."
```

See Section 6 for additional security considerations when handling Display Strings.

4. Working with Structured Fields in HTTP

This section defines how to serialize and parse the abstract types defined by Section 3 into textual HTTP field values and other encodings compatible with them (e.g., in HTTP/2 [HTTP/2] before compression with HPACK [HPACK]).

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_key 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_key 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_key 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_key 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_key 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_key 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. If input_item is a Date, return the result of running Serializing a Date (Section 4.1.10) with input_item.
8. If input_item is a Display String, return the result of running Serializing a Display String (Section 4.1.11) with input_item.
9. 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-ff (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.

4.1.10. Serializing a Date

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

1. Let output be "@".
2. Append to output the result of running Serializing an Integer with input_date (Section 4.1.4).
3. Return output.

4.1.11. Serializing a Display String

Given a sequence of Unicode code points as input_sequence, return an ASCII string suitable for use in an HTTP field value.

1. If input_sequence is not a sequence of Unicode code points, fail serialization.
2. Let byte_array be the result of applying UTF-8 encoding (Section 3 of [UTF8]) to input_sequence. If encoding fails, fail serialization.
3. Let encoded_string be a string containing "%" followed by DQUOTE.
4. For each byte in byte_array:
    1. If byte is %x25 ("%"), %x22 (DQUOTE), or in the ranges %x00-1f or %x7f-ff:
        1. Append "%" to encoded_string.
        2. Let encoded_byte be the result of applying base16 encoding (Section 8 of [RFC4648]) to byte, with any alphabetic characters converted to lowercase.
        3. Append encoded_byte to encoded_string.
    2. Otherwise, decode byte as an ASCII character and append the result to encoded_string.
5. Append DQUOTE to encoded_string.
6. Return encoded_string.

Note that [UTF8] prohibits the encoding of code points between U+D800 and U+DFFF (surrogates); if they occur in input_sequence, serialization will fail.

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 field 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 Section 5.2 of [HTTP]; 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 field 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, either the entire field value **MUST** be ignored (i.e., treated as if the field were not present in the section), or alternatively the complete HTTP message **MUST** be treated as malformed. This is intentionally strict to improve interoperability and safety, and field specifications that use Structured Fields 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. If dictionary already contains a key this_key (comparing character for character), overwrite its value with member.
    5. Otherwise, append key this_key with value member to dictionary.
    6. Discard any leading OWS characters from input_string.
    7. If input_string is empty, return dictionary.
    8. Consume the first character of input_string; if it is not ",", fail parsing.
    9. Discard any leading OWS characters 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).

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. If the first character of input_string is "@", return the result of running Parsing a Date (Section 4.2.9) with input_string.
7. If the first character of input_string is "%", return the result of running Parsing a Display String (Section 4.2.10) with input_string.
8. 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 the ";" character from the beginning of input_string.
    3. Discard any leading SP characters from input_string.
    4. Let param_key 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. If parameters already contains a key param_key (comparing character for character), overwrite its value with param_value.
    8. Otherwise, append key param_key with value param_value to parameters.
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. Let output_number be an Integer that is the result of parsing input_number as an integer.
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. Let output_number be a Decimal that is the result of parsing input_number as a decimal number.
10. Let output_number be the product of output_number and sign.
11. 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-ff (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). If base64 decoding fails, parsing fails.
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 Sections 3.1 and 3.3 of [RFC4648]; 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.

4.2.9. Parsing a Date

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

1. If the first character of input_string is not "@", fail parsing.
2. Discard the first character of input_string.
3. Let output_date be the result of running Parsing an Integer or Decimal (Section 4.2.4) with input_string.
4. If output_date is a Decimal, fail parsing.
5. Return output_date.

4.2.10. Parsing a Display String

Given an ASCII string as input_string, return a sequence of Unicode code points. input_string is modified to remove the parsed value.

1. If the first two characters of input_string are not "%" followed by DQUOTE, fail parsing.
2. Discard the first two characters of input_string.
3. Let byte_array be an empty byte array.
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 in the range %x00-1f or %x7f-ff (i.e., it is not in VCHAR or SP), fail parsing.
    3. If char is "%":
        1. Let octet_hex be the result of consuming two characters from input_string. If there are not two characters, fail parsing.
        2. If octet_hex contains characters outside the range %x30-39 or %x61-66 (i.e., it is not in 0-9 or lowercase a-f), fail parsing.
        3. Let octet be the result of hex decoding octet_hex (Section 8 of [RFC4648]).
        4. Append octet to byte_array.
    4. If char is DQUOTE:
        1. Let unicode_sequence be the result of decoding byte_array as a UTF-8 string (Section 3 of [UTF8]). Fail parsing if decoding fails.
        2. Return unicode_sequence.
    5. Otherwise, if char is not "%" or DQUOTE:
        1. Let byte be the result of applying ASCII encoding to char.
        2. Append byte to byte_array.
5. Reached the end of input_string without finding a closing DQUOTE; fail parsing.

5. IANA Considerations

IANA has added the following note to the "Hypertext Transfer Protocol (HTTP) Field Name Registry":

> The "Structured Type" column indicates the type of the field (per RFC 9651), if any, and may be "Dictionary", "List", or "Item".
>
> Note that field names beginning with characters other than ALPHA or "*" will not be able to be represented as a Structured Fields Token and therefore may be incompatible with being mapped into field values that refer to it.

A new column, "Structured Type", has been added to the registry.

The indicated Structured Type for each existing registry entry listed in Table 1 has also been added.

Field Name	Structured Type
Accept-CH	List
Cache-Status	List
CDN-Cache-Control	Dictionary
Cross-Origin-Embedder-Policy	Item
Cross-Origin-Embedder-Policy-Report-Only	Item
Cross-Origin-Opener-Policy	Item
Cross-Origin-Opener-Policy-Report-Only	Item
Origin-Agent-Cluster	Item
Priority	Dictionary
Proxy-Status	List

Table 1: Existing Fields

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.

The Display String type can convey any possible Unicode code point without sanitization; for example, they might contain unassigned code points, control points (including NUL), or noncharacters. Therefore, applications consuming Display Strings need to consider strategies such as filtering or escaping untrusted content before displaying it. See [PRECIS] and [UNICODE-SECURITY].

7. References

7.1. Normative References

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[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>.

[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>.

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

7.2. Informative References

[HPACK] 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>.

[HTTP/2] Thomson, M., Ed. and C. Benfield, Ed., "HTTP/2", RFC 9113, DOI 10.17487/RFC9113, June 2022, <https://www.rfc-editor.org/info/rfc9113>.

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE Std 754-2019, DOI 10.1109/IEEESTD.2019.8766229, ISBN 978-1-5044-5924-2, July 2019, <https://ieeexplore.ieee.org/document/8766229>.

[PRECIS] Saint-Andre, P. and M. Blanchet, "PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols", RFC 8264, DOI 10.17487/RFC8264, October 2017, <https://www.rfc-editor.org/info/rfc8264>.

[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>.

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

[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>.

[UNICODE-SECURITY] Davis, M. and M. Suignard, "Unicode Security Considerations", Unicode Technical Report #36, 19 September 2014, <https://www.unicode.org/reports/tr36/tr36-15.html>. Latest version available at <https://www.unicode.org/reports/tr36/>.

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 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 built-in 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.

Appendix C. ABNF

This section uses the Augmented Backus-Naur Form (ABNF) notation [RFC5234] to illustrate the expected syntax of Structured Fields. However, it cannot be used to validate their syntax because it does not capture all requirements.

This section is non-normative. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence.

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

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

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

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

sf-item   = bare-item parameters
bare-item = sf-integer / sf-decimal / sf-string / sf-token
            / sf-binary / sf-boolean / sf-date / sf-displaystring

sf-integer       = ["-"] 1*15DIGIT
sf-decimal       = ["-"] 1*12DIGIT "." 1*3DIGIT
sf-string        = DQUOTE *( unescaped / "%" / bs-escaped ) DQUOTE
sf-token         = ( ALPHA / "*" ) *( tchar / ":" / "/" )
sf-binary        = ":" base64 ":"
sf-boolean       = "?" ( "0" / "1" )
sf-date          = "@" sf-integer
sf-displaystring = "%" DQUOTE *( unescaped / "\" / pct-encoded )
                   DQUOTE

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

unescaped    = %x20-21 / %x23-24 / %x26-5B / %x5D-7E
bs-escaped   = "\" ( DQUOTE / "\" )

pct-encoded  = "%" lc-hexdig lc-hexdig
lc-hexdig = DIGIT / %x61-66 ; 0-9, a-f
```

Appendix D. Changes from RFC 8941

This revision of the "Structured Field Values for HTTP" specification has made the following changes:

- Added the Date Structured Type. (Section 3.3.7)
- Stopped encouraging use of ABNF in definitions of new Structured Fields. (Section 2)
- Moved ABNF to an informative appendix. (Appendix C)
- Added a "Structured Type" column to the "Hypertext Transfer Protocol (HTTP) Field Name Registry". (Section 5)
- Refined parse failure handling. (Section 4.2)

- Added the Display String Structured Type. (Section 3.3.8)

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
Cloudflare
Prahran VIC
Australia
Email: mnot@mnot.net
URI: https://www.mnot.net/

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