Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 8, 2017

PH. Kamp
The Varnish Cache Project

October 05, 2016

HTTP header common structure
draft-kamp-httpbis-structure-00

Abstract

An abstract data model for HTTP headers, "Common Structure", and a HTTP/1 serialization of it, generalized from current HTTP headers.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 8, 2017.

1. Introduction

The HTTP protocol does not impose any structure or datamodel on the information in HTTP headers, the HTTP/1 serialization is the datamodel: An ASCII string without control characters.

HTTP header definitions specify how the string must be formatted and while families of similar headers exist, it still requires an uncomfortable large number of bespoke parser and validation routines to process HTTP traffic correctly.

In order to improve performance HTTP/2 and HPACK uses naive text-compression, which incidentally decoupled the on-the-wire serialization from the data model.

During the development of HPACK it became evident that significantly bigger gains were available if semantic compression could be used, most notably with timestamps. However, the lack of a common data structure for HTTP headers would make semantic compression one long list of special cases.

Parallel to this, various proposals for how to fulfill data-transportation needs, and to a lesser degree to impose some kind of order on HTTP headers, at least going forward were floated.

All of these proposals, JSON, CBOR etc. run into the same basic problem: Their serialization is incompatible with [RFC7230]'s ABNF definition of 'field-value'.

For binary formats, such as CBOR, a wholesale base64/85 reserialization would be needed, with negative results for both debugability and bandwidth.

For textual formats, such as JSON, the format must first be neutered to not violate field-value's ABNF, and then workarounds added to reintroduce the features just lost, for instance UNICODE strings, and suddenly it is no longer JSON anymore.

This proposal starts from the other end, and builds and generalizes a data structure definition from existing HTTP headers, which means that HTTP/1 serialization and 'field-value' compatibility is built in.

If all new HTTP headers are defined to fit into this Common Structure we have at least halted the proliferation of bespoke parsers and started to pave the road for semantic compression serializations of HTTP traffic.

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

2. Definition of HTTP header Common Structure

The data model of Common Structure is an ordered sequence of named dictionaries. Please see Appendix A for how this model was derived.

The definition of the data model is on purpose abstract, uncoupled from any protocol serialization or programming environment representation, meant as the foundation on which all such manifestations of the model can be built.

Common Structure in ABNF:

```
import token from RFC7230
import DIGIT from RFC5234

common-structure = 1* ( identifier dictionary )

dictionary = * ( identifier value )

value = identifier /
        number /
        ascii_string /
        unicode_string /
        blob /
        timestamp /
        common-structure

identifier = (token / "*") [ token / "*" ]

number = ["-"] 1*15 DIGIT
        # XXX: Not sure how to do this in ABNF:
        # XXX: A single "." allowed between any two digits
        # The range is limited is to ensure it can be
        # correctly represented in IEEE754 64 bit
        # binary floating point format.

ascii_string = * %x20-7e
        # This is a "safe" string in the sense that it
        # contains no control characters or multi-byte
        # sequences.  If that is not fancy enough, use
        # unicode_string.

unicode_string = * unicode_codepoint
        # XXX: Is there a place to import this from ?
        # Unrestricted unicode, because there is no sane
        # way to restrict or otherwise make unicode "safe".

blob = * %0x00-ff
        # Intended for cryptographic data and as a general
        # escape mechanism for unmet requirements.

timestamp = POSIX time_t with optional millisecond resolution
        # XXX: Is there a place to import this from ?
```

3. HTTP/1 serialization of HTTP header Common Structure

In ABNF:

```
import OWS from {{RFC7230}}
import HEXDIG, DQUOTE from {{RFC5234}}
```

```
h1_common-structure-header =
        ( field-name ":" OWS ">" h1_common_structure "<" )
                # Self-identifying HTTP headers
        ( field-name ":" OWS h1_common_structure ) /
                # legacy HTTP headers on white-list, see {{iana}}

h1_common_structure = h1_element  * ("," h1_element)

h1_element = identifier * (";" identifier ["=" h1_value])

h1_value = identifier /
        number /
        h1_ascii_string /
        h1_unicode_string /
        h1_blob /
        h1_timestamp /
        h1_common-structure

h1_ascii_string = DQUOTE *(
                ( "\" DQUOTE ) /
                ( "\" "\" ) /
                0x20-21 /
                0x23-5B /
                0x5D-7E
                ) DQUOTE
        # This is a proper subset of h1_unicode_string
        # NB only allowed backslash escapes are \" and \\

h1_unicode_string = DQUOTE *(
                ( "\" DQUOTE )
                ( "\" "\" ) /
                ( "\" "u" 4*HEXDIG ) /
                0x20-21 /
                0x23-5B /
                0x5D-7E /
                0x80-F7
                ) DQUOTE
        # XXX: how to say/import "UTF-8 encoding" ?
        # HTTP1 unfriendly codepoints (00-1f, 7f) must be
        # encoded with \uXXXX escapes

h1_blob = "'" base64 "'"
        # XXX: where to import base64 from ?

h1_timestamp = number
        # UNIX/POSIX time_t semantics.
        # fractional seconds allowed.
```

```
h1_common_structure = ">" h1_common_structure "<"
```

XXX: Allow OWS in parsers, but not in generators ?

In programming environments which do not define a native representation or serialization of Common Structure, the HTTP/1 serialization should be used.

4. When to use Common Structure parser

All future standardized and all private HTTP headers using Common Structure should self identify as such. In the HTTP/1 serialization by making the first character ">" and the last "<". (These two characters are deliberately "the wrong way" to not clash with exsisting usages.)

Legacy HTTP headers which fit into Common Structure, are marked as such in the IANA Message Header Registry (see {iana}), and a snapshot of the registry can be used to trigger parsing according to Common Structure of these headers.

5. Desired normative effects

All new HTTP headers SHOULD use the Common Structure if at all possible.

6. Open/Outstanding issues to resolve

6.1. Single/multiple headers

Should we allow splitting common structure data over multiple headers ?

Pro:

Avoids size restrictions, easier on-the-fly editing

Contra:

Cannot act on any such header until all headers have been received.

We must define where headers can be split (between identifier and dictionary ?, in the middle of dictionaries ?)

Most on-the-fly editing is hackish at best.

7. Future work

7.1. Redefining existing headers for better performance

The HTTP/1 serializations self-identification mechanism makes it possible to extend the definition of existing Appendix C headers into Common Structure.

For instance one could imagine:

```
Date: >1475061449.201<
```

Which would be faster to parse and validate than the current definition of the Date header and more precise too.

Some kind of signal/negotiation mechanism would be required to make this work in practice.

7.2. Define a validation dictionary

A machine-readable specification of the legal contents of HTTP headers would go a long way to improve efficiency and security in HTTP implementations.

8. IANA considerations

The IANA Message Header Registry will be extended with an additional field named "Common Structure" which can have the values "True", "False" or "Unknown".

The RFC723x headers listed in Appendix B will get the value "True" in the new field.

The RFC723x headers listed in Appendix C will get the value "False" in the new field.

All other existing entries in the registry will be set to "Unknown" until and if the owner of the entry requests otherwise.

9. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[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, <http://www.rfc-editor.org/info/rfc7230>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <http://www.rfc-editor.org/info/rfc5234>.

Appendix A. Does HTTP headers have any common structure ?

Several proposals have been floated in recent years to use some preexisting structured data serialization or other for HTTP headers, to impose some sanity.

None of these proposals have gained traction and no obvious candidate data serializations have been left unexamined.

This effort tries to tackle the question from the other side, by asking if there is a common structure in existing HTTP headers we can generalize for this purpose.

A.1. Survey of HTTP header structure

The RFC723x family of HTTP/1 standards control 49 entries in the IANA Message Header Registry, and they share two common motifs.

The majority of RFC723x HTTP headers are lists. A few of them are ordered, ('Content-Encoding'), some are unordered ('Connection') and some are ordered by 'q=%f' weight parameters ('Accept')

In most cases, the list elements are some kind of identifier, usually derived from ABNF 'token' as defined by [RFC7230].

A subgroup of headers, mostly related to MIME, uses what one could call a 'qualified token'::

```
qualified_token = token_or_asterix [ "/" token_or_asterix ]
```

The second motif is parameterized list elements. The best known is the "q=0.5" weight parameter, but other parameters exist as well.

Generalizing from these motifs, our candidate "Common Structure" data model becomes an ordered list of named dictionaries.

In pidgin ABNF, ignoring white-space for the sake of clarity, the HTTP/1.1 serialization of Common Structure is is something like:

```
token_or_asterix = token from {{RFC7230}}, but also allowing "*"

qualified_token = token_or_asterix [ "/" token_or_asterix ]

field-name, see {{RFC7230}}

Common_Structure_Header = field-name ":" 1#named_dictionary

named_dictionary = qualified_token [ *(";" param) ]

param = token [ "=" value ]

value = we'll get back to this in a moment.
```

Nineteen out of the RFC723x's 48 headers, almost 40%, can already be parsed using this definition, and none the rest have requirements which could not be met by this data model. See Appendix B and Appendix C for the full survey details.

A.2. Survey of values in HTTP headers

Surveying the datatypes of HTTP headers, standardized as well as private, the following picture emerges:

A.2.1. Numbers

Integer and floating point are both used. Range and precision is mostly unspecified in controlling documents.

Scientific notation (9.192631770e9) does not seem to be used anywhere.

The ranges used seem to be minus several thousand to plus a couple of billions, the high end almost exclusively being POSIX time_t timestamps.

A.2.2. Timestamps

RFC723x text format, but POSIX time_t represented as integer or floating point is not uncommon. ISO8601 have also been spotted.

A.2.3. Strings

The vast majority are pure ASCII strings, with either no escapes, %xx URL-like escapes or C-style back-slash escapes, possibly with the addition of \uxxxx UNICODE escapes.

Where non-ASCII character sets are used, they are almost always implicit, rather than explicit. UTF8 and ISO-8859-1[5] seem to be most common.

A.2.4. Binary blobs

Often used for cryptographic data. Usually in base64 encoding, sometimes ""-quoted more often not. base85 encoding is also seen, usually quoted.

A.2.5. Identifiers

Seems to almost always fit in the RFC723x 'token' definition.

A.3. Is this actually a useful thing to generalize ?

The number one wishlist item seems to be UNICODE strings, with a big side order of not having to write a new parser routine every time somebody comes up with a new header.

Having a common parser would indeed be a good thing, and having an underlying data model which makes it possible define a compressed serialization, rather than rely on serialization to text followed by text compression (ie: HPACK) seems like a good idea too.

However, when using a datamodel and a parser general enough to transport useful data, it will have to be followed by a validation step, which checks that the data also makes sense.

Today validation, such as it is, is often done by the bespoke parsers.

This then is probably where the next big potential for improvement lies:

Ideally a machine readable "data dictionary" which makes it possibly to copy that text out of RFCs, run it through a code generator which spits out validation code which operates on the output of the common parser.

But history has been particularly unkind to that idea.

Most attempts studied as part of this effort, have sunk under complexity caused by reaching for generality, but where scope has been wisely limited, it seems to be possible.

So file that idea under "future work".

Appendix B. RFC723x headers with "common structure"

Accept	[RFC7231, Section 5.3.2]
Accept-Charset	[RFC7231, Section 5.3.3]
Accept-Encoding	[RFC7231, Section 5.3.4][RFC7694, Section 3]
Accept-Language	[RFC7231, Section 5.3.5]
Age	[RFC7234, Section 5.1]
Allow	[RFC7231, Section 7.4.1]
Connection	[RFC7230, Section 6.1]
Content-Encoding	[RFC7231, Section 3.1.2.2]
Content-Language	[RFC7231, Section 3.1.3.2]
Content-Length	[RFC7230, Section 3.3.2]
Content-Type	[RFC7231, Section 3.1.1.5]
Expect	[RFC7231, Section 5.1.1]
Max-Forwards	[RFC7231, Section 5.1.2]
MIME-Version	[RFC7231, Appendix A.1]
TE	[RFC7230, Section 4.3]
Trailer	[RFC7230, Section 4.4]
Transfer-Encoding	[RFC7230, Section 3.3.1]
Upgrade	[RFC7230, Section 6.7]
Vary	[RFC7231, Section 7.1.4]

Appendix C. RFC723x headers with "uncommon structure"

1 of the RFC723x headers is only reserved, and therefore have no structure at all:

Close	[RFC7230, Section 8.1]

5 of the RFC723x headers are HTTP dates:

Date	[RFC7231, Section 7.1.1.2]
Expires	[RFC7234, Section 5.3]
If-Modified-Since	[RFC7232, Section 3.3]
If-Unmodified-Since	[RFC7232, Section 3.4]
Last-Modified	[RFC7232, Section 2.2]

24 of the RFC723x headers use bespoke formats which only a single or in rare cases two headers share:

Accept-Ranges	[RFC7233, Section 2.3]
	bytes-unit / other-range-unit
Authorization	[RFC7235, Section 4.2]
Proxy-Authorization	[RFC7235, Section 4.4]
	credentials
Cache-Control	[RFC7234, Section 5.2]

```
        1#cache-directive

    Content-Location    [RFC7231, Section 3.1.4.2]
        absolute-URI / partial-URI

    Content-Range       [RFC7233, Section 4.2]
        byte-content-range / other-content-range

    ETag                [RFC7232, Section 2.3]
        entity-tag

    Forwarded           [RFC7239]
        1#forwarded-element

    From                [RFC7231, Section 5.5.1]
        mailbox

    If-Match            [RFC7232, Section 3.1]
    If-None-Match       [RFC7232, Section 3.2]
        "*" / 1#entity-tag

    If-Range            [RFC7233, Section 3.2]
        entity-tag / HTTP-date

    Host                [RFC7230, Section 5.4]
        uri-host [ ":" port ]

    Location            [RFC7231, Section 7.1.2]
        URI-reference

    Pragma              [RFC7234, Section 5.4]
        1#pragma-directive

    Range               [RFC7233, Section 3.1]
        byte-ranges-specifier / other-ranges-specifier

    Referer             [RFC7231, Section 5.5.2]
        absolute-URI / partial-URI

    Retry-After         [RFC7231, Section 7.1.3]
        HTTP-date / delay-seconds

    Server              [RFC7231, Section 7.4.2]
    User-Agent          [RFC7231, Section 5.5.3]
        product *( RWS ( product / comment ) )

    Via                 [RFC7230, Section 5.7.1]
        1#( received-protocol RWS received-by [ RWS comment ] )
```

```
       Warning             [RFC7234, Section 5.5]
           1#warning-value

       Proxy-Authenticate  [RFC7235, Section 4.3]
       WWW-Authenticate    [RFC7235, Section 4.1]
           1#challenge
```

Author's Address

Poul-Henning Kamp
The Varnish Cache Project

Email: phk@varnish-cache.org