

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2018

H. Birkholz
Fraunhofer SIT
C. Vigano
Universitaet Bremen
C. Bormann
Universitaet Bremen TZI
July 03, 2017

Concise data definition language (CDDL): a notational convention to
express CBOR data structures
draft-greevenbosch-appsawg-cbor-cddl-11

Abstract

This document proposes a notational convention to express CBOR data structures (RFC 7049). Its main goal is to provide an easy and unambiguous way to express structures for protocol messages and data formats that use CBOR.

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 January 4, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements notation	4
1.2. Terminology	4
2. The Style of Data Structure Specification	4
2.1. Groups and Composition in CDDL	6
2.1.1. Usage	8
2.1.2. Syntax	8
2.2. Types	8
2.2.1. Values	9
2.2.2. Choices	9
2.2.3. Representation Types	10
2.2.4. Root type	11
3. Syntax	11
3.1. General conventions	11
3.2. Occurrence	13
3.3. Predefined names for types	14
3.4. Arrays	14
3.5. Maps	15
3.5.1. Structs	15
3.5.2. Tables	18
3.6. Tags	19
3.7. Unwrapping	19
3.8. Controls	20
3.8.1. Control operator .size	21
3.8.2. Control operator .bits	21
3.8.3. Control operator .regexp	22
3.8.4. Control operators .cbor and .cborseq	23
3.8.5. Control operators .within and .and	23
3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default	24
3.9. Socket/Plug	24
3.10. Generics	26
3.11. Operator Precedence	26
4. Making Use of CDDL	28
4.1. As a guide to a human user	28
4.2. For automated checking of CBOR data structure	28
4.3. For data analysis tools	29
5. Security considerations	29
6. IANA considerations	29
7. Acknowledgements	30
8. References	30
8.1. Normative References	30

8.2. Informative References	31
Appendix A. Cemetery	31
A.1. Resolved Issues	32
Appendix B. (Not used.)	32
Appendix C. Change Log	32
Appendix D. ABNF grammar	35
Appendix E. Standard Prelude	37
E.1. Use with JSON	39
Appendix F. The CDDL tool	41
Appendix G. Extended Diagnostic Notation	41
G.1. White space in byte string notation	42
G.2. Text in byte string notation	42
G.3. Embedded CBOR and CBOR sequences in byte strings	42
G.4. Concatenated Strings	43
G.5. Hexadecimal, octal, and binary numbers	43
G.6. Comments	44
Appendix H. Examples	44
H.1. RFC 7071	45
H.1.1. Examples from JSON Content Rules	48
Authors' Addresses	50

1. Introduction

In this document, a notational convention to express CBOR [RFC7049] data structures is defined.

The main goal for the convention is to provide a unified notation that can be used when defining protocols that use CBOR. We term the convention "Concise data definition language", or CDDL.

The CBOR notational convention has the following goals:

- (G1) Provide an unambiguous description of the overall structure of a CBOR data structure.
- (G2) Flexibility to express the freedoms of choice in the CBOR data format.
- (G3) Possibility to restrict format choices where appropriate [_format].
- (G4) Able to express common CBOR datatypes and structures.
- (G5) Human and machine readable and processable.
- (G6) Automatic checking of data format compliance.

(G7) Extraction of specific elements from CBOR data for further processing.

Not an explicit goal per se, but a convenient side effect of the JSON generic data model being a subset of the CBOR generic data model, is the fact that CDDL can also be used for describing JSON data structures (see Appendix E.1).

This document has the following structure:

The syntax of CDDL is defined in Section 3. Examples of CDDL and related CBOR data instances are defined in Appendix H. Section 4 discusses usage of CDDL. Examples are provided early in the text to better illustrate concept definitions. A formal definition of CDDL using ABNF grammar is provided in Appendix D. Finally, a prelude of standard CDDL definitions available in every CBOR specification is listed in Appendix E.

1.1. Requirements notation

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 RFC 2119, BCP 14 [RFC2119].

1.2. Terminology

New terms are introduced in *_cursive_*. CDDL text in the running text is in "typewriter".

2. The Style of Data Structure Specification

CDDL focuses on styles of specification that are in use in the community employing the data model as pioneered by JSON and now refined in CBOR.

There are a number of more or less atomic elements of a CBOR data model, such as numbers, simple values (false, true, nil), text and byte strings; CDDL does not focus on specifying their structure. CDDL of course also allows adding a CBOR tag to a data item.

The more important components of a data structure definition language are the data types used for composition: arrays and maps in CBOR (called arrays and objects in JSON). While these are only two representation formats, they are used to specify four loosely distinguishable styles of composition:

- o A `_vector_`, an array of elements that are mostly of the same semantics. The set of signatures associated with a signed data item is a typical application of a vector.
- o A `_record_`, an array the elements of which have different, positionally defined semantics, as detailed in the data structure definition. A 2D point, specified as an array of an x coordinate (which comes first) and a y coordinate (coming second) is an example of a record, as is the pair of exponent (first) and mantissa (second) in a CBOR decimal fraction.
- o A `_table_`, a map from a domain of map keys to a domain of map values, that are mostly of the same semantics. A set of language tags, each mapped to a text string translated to that specific language, is an example of a table. The key domain is usually not limited to a specific set by the specification, but open for the application, e.g., in a table mapping IP addresses to MAC addresses, the specification does not attempt to foresee all possible IP addresses.
- o A `_struct_`, a map from a domain of map keys as defined by the specification to a domain of map values the semantics of each of which is bound to a specific map key. This is what many people have in mind when they think about JSON objects; CBOR adds the ability to use map keys that are not just text strings. Structs can be used to solve similar problems as records; the use of explicit map keys facilitates optionality and extensibility.

Two important concepts provide the foundation for CDDL:

1. Instead of defining all four types of composition in CDDL separately, or even defining one kind for arrays (vectors and records) and one kind for maps (tables and structs), there is only one kind of composition in CDDL: the `_group_` (Section 2.1).
2. The other important concept is that of a `_type_`. The entire CDDL specification defines a type (the one defined by its first `_rule_`), which formally is the set of CBOR instances that are acceptable for this specification. CDDL predefines a number of basic types such as "uint" (unsigned integer) or "tstr" (text string), often making use of a simple formal notation for CBOR data items. Each value that can be expressed as a CBOR data item also is a type in its own right, e.g. "1". A type can be built as a `_choice_` of other types, e.g., an "int" is either a "uint" or a "nint" (negative integer). Finally, a type can be built as an array or a map from a group.

2.1. Groups and Composition in CDDL

CDDL Groups are lists of name/value pairs (group `_entries_`).

In an array context, only the value of the entry is represented; the name is annotation only (and can be left off if not needed). In a map context, the names become the map keys ("member keys").

In an array context, the sequence of elements in the group is important, as it is the information that allows associating actual array elements with entries in the group. In a map context, the sequence of entries in a group is not relevant (but there is still a need to write down group entries in a sequence).

A group can be placed in (round) parentheses, and given a name by using it in a rule:

```
pii = (  
    age: int,  
    name: tstr,  
    employer: tstr,  
)
```

Figure 1: A basic group

Or a group can just be used in the definition of something else:

```
person = {(  
    age: int,  
    name: tstr,  
    employer: tstr,  
)}
```

Figure 2: Using a group in a map

which, given the above rule for `pii`, is identical to:

```
person = {  
    pii  
}
```

Figure 3: Using a group by name

Note that the (curly) braces signify the creation of a map; the groups themselves are neutral as to whether they will be used in a map or an array.

The parentheses for groups are optional when there is some other set of brackets present, so it would be slightly more natural to express Figure 2 as:

```
person = {  
    age: int,  
    name: tstr,  
    employer: tstr,  
}
```

Groups can be used to factor out common parts of structs, e.g., instead of writing:

```
person = {  
    age: int,  
    name: tstr,  
    employer: tstr,  
}  
  
dog = {  
    age: int,  
    name: tstr,  
    leash-length: float,  
}
```

one can choose a name for the common subgroup and write:

```
person = {  
    identity,  
    employer: tstr,  
}  
  
dog = {  
    identity,  
    leash-length: float,  
}  
  
identity = (  
    age: int,  
    name: tstr,  
)
```

Figure 4: Using a group for factorization

Note that the contents of the braces in the above definitions constitute (anonymous) groups, while "identity" is a named group.

2.1.1. Usage

Groups are the instrument used in composing data structures with CDDL. It is a matter of style in defining those structures whether to define groups (anonymously) right in their contexts or whether to define them in a separate rule and to reference them with their respective name (possibly more than once).

With this, one is allowed to define all small parts of their data structures and compose bigger protocol units with those or to have only one big protocol data unit that has all definitions ad hoc where needed.

2.1.2. Syntax

The composition syntax intends to be concise and easy to read:

- o The start of a group can be marked by '('
- o The end of a group can be marked by ')'
- o Definitions of entries inside of a group are noted as follows: `_keytype => valuetype, _` (read "keytype maps to valuetype"). The comma is actually optional (not just in the final entry), but it is considered good style to set it. The double arrow can be replaced by a colon in the common case of directly using a text string as a key (see Section 3.5.1).

An entry consists of a `_keytype_` and a `_valuetype_`:

- o `_keytype_` is either an atom used as the actual key or a type in general. The latter case may be needed when using groups in a table context, where the actual keys are of lesser importance than the key types, e.g. in contexts verifying incoming data.
- o `_valuetype_` is a type, which could be derived from the major types defined in [RFC7049], could be a convenience valuetype defined in this document (Appendix E) or the name of a type defined in the specification.

A group definition can also contain choices between groups, see Section 2.2.2.

2.2. Types

2.2.1. Values

Values such as numbers and strings can be used in place of a type. (For instance, this is a very common thing to do for a keytype, common enough that CDDL provides additional convenience syntax for this.)

2.2.2. Choices

Many places that allow a type also allow a choice between types, delimited by a "/" (slash). The entire choice construct can be put into parentheses if this is required to make the construction unambiguous (please see Appendix D for the details).

Choices of values can be used to express enumerations:

```
attire = "bow tie" / "necktie" / "Internet attire"
protocol = 6 / 17
```

Similarly as for types, CDDL also allows choices between groups, delimited by a "/" (double slash).

```
address = { delivery }

delivery = (
  street: tstr, ? number: uint, city //
  po-box: uint, city //
  per-pickup: true )

city = (
  name: tstr, zip-code: uint
)
```

Both for type choices and for group choices, additional alternatives can be added to a rule later in separate rules by using "/" and "//=", respectively, instead of "=":

```
attire /= "swimwear"

delivery // = (
  lat: float, long: float, drone-type: tstr
)
```

It is not an error if a name is first used with a "/" or "//=" (there is no need to "create it" with "=").

2.2.2.1. Ranges

Instead of naming all the values that make up a choice, CDDL allows building a `_range_` out of two values that are in an ordering relationship. A range can be inclusive of both ends given (denoted by joining two values by `".."`), or include the first and exclude the second (denoted by instead using `"..."`).

```
device-address = byte
max-byte = 255
byte = 0..max-byte ; inclusive range
first-non-byte = 256
byte1 = 0...first-non-byte ; byte1 is equivalent to byte
```

CDDL currently only allows ranges between numbers [`_range_`].

2.2.2.2. Turning a group into a choice

Some choices are built out of large numbers of values, often integers, each of which is best given a semantic name in the specification. Instead of naming each of these integers and then accumulating these into a choice, CDDL allows building a choice from a group by prefixing it with a `"&"` character:

```
terminal-color = &basecolors
basecolors = (
  black: 0, red: 1, green: 2, yellow: 3,
  blue: 4, magenta: 5, cyan: 6, white: 7,
)
extended-color = &(
  basecolors,
  orange: 8, pink: 9, purple: 10, brown: 11,
)
```

As with the use of groups in arrays (Section 3.4), the membernames have only documentary value (in particular, they might be used by a tool when displaying integers that are taken from that choice).

2.2.3. Representation Types

CDDL allows the specification of a data item type by referring to the CBOR representation (major and minor numbers). How this is used should be evident from the prelude (Appendix E).

It may be necessary to make use of representation types outside the prelude, e.g., a specification could start by making use of an existing tag in a more specific way, or define a new tag not defined in the prelude:

```
my_breakfast = #6.55799(breakfast)    ; cbor-any is too general!
breakfast = cereal / porridge
cereal = #6.998(tstr)
porridge = #6.999([liquid, solid])
liquid = milk / water
milk = 0
water = 1
solid = tstr
```

2.2.4. Root type

There is no special syntax to identify the root of a CDDL data structure definition: that role is simply taken by the first rule defined in the file.

This is motivated by the usual top-down approach for defining data structures, decomposing a big data structure unit into smaller parts; however, except for the root type, there is no need to strictly follow this sequence.

(Note that there is no way to use a group as a root - it must be a type. Using a group as the root might be employed as a way to specify a CBOR sequence in a future version of this specification; this would act as if that group is used in an array and the data items in that fictional array form the members of the CBOR sequence.)

3. Syntax

In this section, the overall syntax of CDDL is shown, alongside some examples just illustrating syntax. (The definition will not attempt to be overly formal; refer to Appendix D for the details.)

3.1. General conventions

The basic syntax is inspired by ABNF [RFC5234], with

- o rules, whether they define groups or types, are defined with a name, followed by an equals sign "=" and the actual definition according to the respective syntactic rules of that definition.
- o A name can consist of any of the characters from the set {'A', ..., 'Z', 'a', ..., 'z', '0', ..., '9', '_', '-', '@', '.', '\$'}, starting with an alphabetic character (including '@', '_', '\$') and ending in one or a digit.
 - * Names are case sensitive.
 - * It is preferred style to start a name with a lower case letter.

- * The hyphen is preferred over the underscore (except in a "bareword" (Section 3.5.1), where the semantics may actually require an underscore).
- * The period may be useful for larger specifications, to express some module structure (as in "tcp.throughput" vs. "udp.throughput").
- * A number of names are predefined in the CDDL prelude, as listed in Appendix E.
- * Rule names (types or groups) do not appear in the actual CBOR encoding, but names used as "barewords" in member keys do.
- o Comments are started by a ';' (semicolon) character and finish at the end of a line (LF or CRLF).
- o outside strings, whitespace (spaces, newlines, and comments) is used to separate syntactic elements for readability (and to separate identifiers or numbers that follow each other); it is otherwise completely optional.
- o Hexadecimal numbers are preceded by '0x' (without quotes, lower case x), and are case insensitive. Similarly, binary numbers are preceded by '0b'.
- o Text strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in section 7 of [RFC7159]. (ABNF users may want to note that there is no support in CDDL for the concept of case insensitivity in text strings; if necessary, regular expressions can be used (Section 3.8.3).)
- o Byte strings are enclosed by single quotation "'" characters and may be prefixed by "h" or "b64". If unprefixed, the string is interpreted as with a text string, except that single quotes must be escaped and that the UTF-8 bytes resulting are marked as a byte string (major type 2). If prefixed as "h" or "b64", the string is interpreted as a sequence of hex digits or a base64(url) string, respectively (as with the diagnostic notation in section 6 of [RFC7049]; cf. Appendix G.2); any white space present within the string (including comments) is ignored in the prefixed case.
[_strings]
- o CDDL uses UTF-8 [RFC3629] for its encoding.

Example:

```
; This is a comment
person = { g }

g = (
  "name": tstr,
  age: int, ; "age" is a bareword
)
```

3.2. Occurrence

An optional `_occurrence_` indicator can be given in front of a group entry. It is either one of the characters `'?'` (optional), `'*'` (zero or more), or `'+'` (one or more), or is of the form `n*m`, where `n` and `m` are optional unsigned integers and `n` is the lower limit (default 0) and `m` is the upper limit (default no limit) of occurrences.

If no occurrence indicator is specified, the group entry is to occur exactly once (as if `1*1` were specified).

Note that CDDL, outside any directives/annotations that could possibly be defined, does not make any prescription as to whether arrays or maps use the definite length or indefinite length encoding. I.e., there is no correlation between leaving the size of an array "open" in the spec and the fact that it is then interchanged with definite or indefinite length.

Please also note that CDDL can describe flexibility that the data model of the target representation does not have. This is rather obvious for JSON, but also is relevant for CBOR:

```
apartment = {
  kitchen: size,
  * bedroom: size,
}
size = float ; in m2
```

The previous specification does not mean that CBOR is changed to allow to use the key "bedroom" more than once. In other words, due to the restrictions imposed by the data model, the third line pretty much turns into:

```
? bedroom: size,
```

(Occurrence indicators beyond one still are useful in maps for groups that allow a variety of keys.)

3.3. Predefined names for types

CDDL predefines a number of names. This subsection summarizes these names, but please see Appendix E for the exact definitions.

The following keywords for primitive datatypes are defined:

"bool" Boolean value (major type 7, additional information 20 or 21).

"uint" An unsigned integer (major type 0).

"nint" A negative integer (major type 1).

"int" An unsigned integer or a negative integer.

"float16" IEEE 754 half-precision float (major type 7, additional information 25).

"float32" IEEE 754 single-precision float (major type 7, additional information 26).

"float64" IEEE 754 double-precision float (major type 7, additional information 27).

"float" One of float16, float32, or float64.

"bstr" or "bytes" A byte string (major type 2).

"tstr" or "text" Text string (major type 3)

(Note that there are no predefined names for arrays or maps; these are defined with the syntax given below.)

In addition, a number of types are defined in the prelude that are associated with CBOR tags, such as "tdate", "bigint", "regexp" etc.

3.4. Arrays

Array definitions surround a group with square brackets.

For each entry, an occurrence indicator as specified in Section 3.2 is permitted.

For example:

```
unlimited-people = [* person]
one-or-two-people = [1*2 person]
at-least-two-people = [2* person]
person = (
    name: tstr,
    age: uint,
)
```

The group "person" is defined in such a way that repeating it in the array each time generates alternating names and ages, so these are four valid values for a data item of type "unlimited-people":

```
["roundlet", 1047, "psychurgy", 2204, "extrarhythmical", 2231]
[]
["aluminize", 212, "climograph", 4124]
["penintime", 1513, "endocarditis", 4084, "impermeator", 1669,
 "coextension", 865]
```

3.5. Maps

The syntax for specifying maps merits special attention, as well as a number of optimizations and conveniences, as it is likely to be the focal point of many specifications employing CDDL. While the syntax does not strictly distinguish struct and table usage of maps, it caters specifically to each of them.

3.5.1. Structs

The "struct" usage of maps is similar to the way JSON objects are used in many JSON applications.

A map is defined in the same way as defining an array (see Section 3.4), except for using curly braces "{}" instead of square brackets "["].

An occurrence indicator as specified in Section 3.2 is permitted for each group entry.

The following is an example of a structure:

```
Geography = [  
  city      : tstr,  
  gpsCoordinates : GpsCoordinates,  
]  
  
GpsCoordinates = {  
  longitude : uint,          ; multiplied by 10^7  
  latitude  : uint,          ; multiplied by 10^7  
}
```

When encoding, the Geography structure is encoded using a CBOR array with two entries (the keys for the group entries are ignored), whereas the GpsCoordinates are encoded as a CBOR map with two key/value pairs.

Types used in a structure can be defined in separate rules or just in place (potentially placed inside parentheses, such as for choices). E.g.:

```
located-samples = {  
  sample-point: int,  
  samples: [+ float],  
}
```

where "located-samples" is the datatype to be used when referring to the struct, and "sample-point" and "samples" are the keys to be used. This is actually a complete example: an identifier that is followed by a colon can be directly used as the text string for a member key (we speak of a "bareword" member key), as can a double-quoted string or a number. (When other types, in particular multi-valued ones, are used as keytypes, they are followed by a double arrow, see below.)

If a text string key does not match the syntax for an identifier (or if the specifier just happens to prefer using double quotes), the text string syntax can also be used in the member key position, followed by a colon. The above example could therefore have been written with quoted strings in the member key positions.

All the types defined can be used in a keytype position by following them with a double arrow. A string also is a (single-valued) type, so another form for this example is:

```
located-samples = {  
  "sample-point" => int,  
  "samples" => [+ float],  
}
```


A better way to demonstrate the double-arrow use may be:

```
located-samples = {  
    sample-point: int,  
    samples: [+ float],  
    * equipment-type => equipment-tolerances,  
}  
equipment-type = [name: tstr, manufacturer: tstr]  
equipment-tolerances = [+ [float, float]]
```

The example below defines a struct with optional entries: display name (as a text string), the name components first name and family name (as a map of text strings), and age information (as an unsigned integer).

```
PersonalData = {  
    ? displayName: tstr,  
    NameComponents,  
    ? age: uint,  
}  
  
NameComponents = (  
    ? firstName: tstr,  
    ? familyName: tstr,  
)
```

Note that the group definition for NameComponents does not generate another map; instead, all four keys are directly in the struct built by PersonalData.

In this example, all key/value pairs are optional from the perspective of CDDL. With no occurrence indicator, an entry is mandatory.

If the addition of more entries not specified by the current specification is desired, one can add this possibility explicitly:

```
PersonalData = {  
  ? displayName: tstr,  
  NameComponents,  
  ? age: uint,  
  * tstr => any  
}  
  
NameComponents = (  
  ? firstName: tstr,  
  ? familyName: tstr,  
)
```

Figure 5: Personal Data: Example for extensibility

The cddl tool (Appendix F) generated as one acceptable instance for this specification:

```
{"familyName": "agust", "antiforeignism": "pretzel",  
 "springbuck": "illuminatingly", "exuviae": "ephemeris",  
 "kilometrage": "frogfish"}
```

(See Section 3.9 for one way to explicitly identify an extension point.)

3.5.2. Tables

A table can be specified by defining a map with entries where the keytype is not single-valued, e.g.:

```
square-roots = { * x => y }  
x = int  
y = float
```

Here, the key in each key/value pair has datatype *x* (defined as int), and the value has datatype *y* (defined as float).

If the specification does not need to restrict one of *x* or *y* (i.e., the application is free to choose per entry), it can be replaced by the predefined name "any".

As another example, the following could be used as a conversion table converting from an integer or float to a string:

```
tostring = { * mynumber => tstr }  
mynumber = int / float
```

3.6. Tags

A type can make use of a CBOR tag (major type 6) by using the representation type notation, giving `#6.nnn(type)` where `nnn` is an unsigned integer giving the tag number and `"type"` is the type of the data item being tagged.

For example, the following line from the CDDL prelude (Appendix E) defines `"biguint"` as a type name for a positive bignum `N`:

```
biguint = #6.2(bstr)
```

The tags defined by [RFC7049] are included in the prelude. Additional tags since registered need to be added to a CDDL specification as needed; e.g., a binary UUID tag could be referenced as `"buuid"` in a specification after defining

```
buuid = #6.37(bstr)
```

In the following example, usage of the tag 32 for URIs is optional:

```
my_uri = #6.32(tstr) / tstr
```

3.7. Unwrapping

The group that is used to define a map or an array can often be reused in the definition of another map or array. Similarly, a type defined as a tag carries an internal data item that one would like to refer to. In these cases, it is expedient to simply use the name of the map, array, or tag type as a handle for the group or type defined inside it.

The `"unwrap"` operator (written by preceding a name by a tilde character `"~"`) can be used to strip the type defined for a name by one layer, exposing the underlying group (for maps and arrays) or type (for tags).

For example, an application might want to define a basic and an advanced header. Without unwrapping, this might be done as follows:

```
basic-header-group = (  
    field1: int,  
    field2: text,  
)  
  
basic-header = { basic-header-group }  
  
advanced-header = {  
    basic-header-group,  
    field3: bytes,  
    field4: number, ; as in the tagged type "time"  
}
```

Unwrapping simplifies this to:

```
basic-header = {  
    field1: int,  
    field2: text,  
}  
  
advanced-header = {  
    ~basic-header,  
    field3: bytes,  
    field4: ~time,  
}
```

(Note that leaving out the first unwrap operator in the latter example would lead to nesting the basic-header in its own map inside the advanced-header, while, with the unwrapped basic-header, the definition of the group inside basic-header is essentially repeated inside advanced-header, leading to a single map. This can be used for various applications often solved by inheritance in programming languages. The effect of unwrapping can also be described as "threading in" the group or type inside the referenced type, which suggested the thread-like "~" character.)

3.8. Controls

A `_control_` allows to relate a `_target_` type with a `_controller_` type via a `_control operator_`.

The syntax for a control type is "target .control-operator controller", where control operators are special identifiers prefixed by a dot. (Note that `_target_` or `_controller_` might need to be parenthesized.)

A number of control operators are defined at this point. Note that the CDDL tool does not currently support combining multiple controls on a single target.

3.8.1. Control operator `.size`

A `".size"` control controls the size of the target in bytes by the control type. Examples:

```
full-address = [[+ label], ip4, ip6]
ip4 = bstr .size 4
ip6 = bstr .size 16
label = bstr .size (1..63)
```

Figure 6: Control for size in bytes

When applied to an unsigned integer, the `".size"` control restricts the range of that integer by giving a maximum number of bytes that should be needed in a computer representation of that unsigned integer. In other words, `"uint .size N"` is equivalent to `"0...BYTES_N"`, where `BYTES_N == 256*N`.

```
audio_sample = uint .size 3 ; 24-bit, equivalent to 0..16777215
```

Figure 7: Control for integer size in bytes

Note that, as with value restrictions in CDDL, this control is not a representation constraint; a number that fits into fewer bytes can still be represented in that form, and an inefficient implementation could use a longer form (unless that is restricted by some format constraints outside of CDDL, such as the rules in Section 3.9 of [RFC7049]).

3.8.2. Control operator `.bits`

A `".bits"` control on a byte string indicates that, in the target, only the bits numbered by a number in the control type are allowed to be set. (Bits are counted the usual way, bit number `"n"` being set in `"str"` meaning that `"(str[n >> 3] & (1 << (n & 7))) != 0"`.)
[`_bitsendian`]

Similarly, a `".bits"` control on an unsigned integer `"i"` indicates that for all unsigned integers `"n"` where `"(i & (1 << n)) != 0"`, `"n"` must be in the control type.

```

tcpflagbytes = bstr .bits flags
flags = &(amp;
    fin: 8,
    syn: 9,
    rst: 10,
    psh: 11,
    ack: 12,
    urg: 13,
    ece: 14,
    cwr: 15,
    ns: 0,
) / (4..7) ; data offset bits

rxwbits = uint .bits rxw
rxw = &(r: 2, w: 1, x: 0)

```

Figure 8: Control for what bits can be set

The CDDL tool generates the following ten example instances for "tcpflagbytes":

```

h'906d' h'01fc' h'8145' h'01b7' h'013d' h'409f' h'018e' h'c05f'
h'01fa' h'01fe'

```

These examples do not illustrate that the above CDDL specification does not explicitly specify a size of two bytes: A valid all clear instance of flag bytes could be "h'" or "h'00'" or even "h'000000'" as well.

3.8.3. Control operator .regexp

A ".regexp" control indicates that the text string given as a target needs to match the PCRE regular expression given as a value in the control type, where that regular expression is anchored on both sides. (If anchoring is not desired for a side, "." needs to be inserted there.)

```

nai = tstr .regexp "\\w+@\\w+(\\.\\.\\w+)+"

```

Figure 9: Control with a PCRE regexp

The CDDL tool proposes:

```

"N1@CH57HF.4Znqe0.dYJRN.igjf"

```

3.8.4. Control operators .cbor and .cborseq

A ".cbor" control on a byte string indicates that the byte string carries a CBOR encoded data item. Decoded, the data item matches the type given as the right-hand side argument (type1 in the following example).

```
"bytes .cbor type1"
```

Similarly, a ".cborseq" control on a byte string indicates that the byte string carries a sequence of CBOR encoded data items. When the data items are taken as an array, the array matches the type given as the right-hand side argument (type2 in the following example).

```
"bytes .cborseq type2"
```

(The conversion of the encoded sequence to an array can be effected for instance by wrapping the byte string between the two bytes 0x9f and 0xff and decoding the wrapped byte string as a CBOR encoded data item.)

3.8.5. Control operators .within and .and

A ".and" control on a type indicates that the data item matches both that left hand side type and the type given as the right hand side. (Formally, the resulting type is the intersection of the two types given.)

```
"type1 .and type2"
```

A variant of the ".and" control is the ".within" control, which expresses an additional intent: the left hand side type is meant to be a subset of the right-hand-side type.

```
"type1 .within type2"
```

While both forms have the identical formal semantics (intersection), the intention of the ".within" form is that the right hand side gives guidance to the types allowed on the left hand side, which typically is a socket (Section 3.9):

```
message = $message .within message-structure
message-structure = [message_type, *message_option]
message_type = 0..255
message_option = any

$message /= [3, dough: text, topping: [* text]]
$message /= [4, noodles: text, sauce: text, parmesan: bool]
```

For ".within", a tool might flag an error if type1 allows data items that are not allowed by type2. In contrast, for ".and", there is no expectation that type1 already is a subset of type2.

3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default

The controls .lt, .le, .gt, .ge, .eq, .ne specify a constraint on the left hand side type to be a value less than, less than or equal, equal to, not equal to, greater than, or greater than or equal to a value given as a (single-valued) right hand side type. In the present specification, the first four controls (.lt, .le, .gt, .ge) are defined only for numeric types, as these have a natural ordering relationship.

```
speed = number .ge 0 ; unit: m/s
```

A variant of the ".ne" control is the ".default" control, which expresses an additional intent: the value specified by the right-hand-side type is intended as a default value for the left hand side type given, and the implied .ne control is there to prevent this value from being sent over the wire. This control is only meaningful when the control type is used in an optional context; otherwise there would be no way to express the default value.

```
timer = {  
  time: uint,  
  ? displayed-step: (number .gt 0) .default 1  
}
```

3.9. Socket/Plug

Both for type choices and group choices, a mechanism is defined that facilitates starting out with empty choices and assembling them later, potentially in separate files that are concatenated to build the full specification.

Per convention, CDDL extension points are marked with a leading dollar sign (types) or two leading dollar signs (groups). Tools honor that convention by not raising an error if such a type or group is not defined at all; the symbol is then taken to be an empty type choice (group choice), i.e., no choice is available.


```
tcp-header = {seq: uint, ack: uint, * $$tcp-option}  
  
; later, in a different file  
  
$$tcp-option /= (  
  sack: [(left: uint, right: uint)]  
)  
  
; and, maybe in another file  
  
$$tcp-option /= (  
  sack-permitted: true  
)
```

Names that start with a single "\$" are "type sockets", names with a double "\$\$" are "group sockets". It is not an error if there is no definition for a socket at all; this then means there is no way to satisfy the rule (i.e., the choice is empty).

All definitions (plugs) for socket names must be augmentations, i.e., they must be using "/"= and "//=", respectively.

To pick up the example illustrated in Figure 5, the socket/plug mechanism could be used as shown in Figure 10:

```
PersonalData = {  
    ? displayName: tstr,  
    NameComponents,  
    ? age: uint,  
    * $$personaldata-extensions  
}  
  
NameComponents = (  
    ? firstName: tstr,  
    ? familyName: tstr,  
)  
  
; The above already works as is.  
; But then, we can add later:  
  
$$personaldata-extensions //= (  
    favorite-salsa: tstr,  
)  
  
; and again, somewhere else:  
  
$$personaldata-extensions //= (  
    shoesize: uint,  
)
```

Figure 10: Personal Data example: Using socket/plugin extensibility

3.10. Generics

Using angle brackets, the left hand side of a rule can add formal parameters after the name being defined, as in:

```
messages = message<"reboot", "now"> / message<"sleep", 1..100>  
message<t, v> = {type: t, value: v}
```

When using a generic rule, the formal parameters are bound to the actual arguments supplied (also using angle brackets), within the scope of the generic rule (as if there were a rule of the form `parameter = argument`).

(There are some limitations to nesting of generics in Appendix F at this time.)

3.11. Operator Precedence

As with any language that has multiple syntactic features such as prefix and infix operators, CDDL has operators that bind more tightly than others. This is becoming more complicated than, say, in ABNF,

as CDDL has both types and groups, with operators that are specific to these concepts. Type operators (such as "/" for type choice) operate on types, while group operators (such as "//" for group choice) operate on groups. Types can simply be used in groups, but groups need to be bracketed (as arrays or maps) to become types. So, type operators naturally bind closer than group operators.

For instance, in

```
t = [group1]
group1 = (a / b // c / d)
a = 1 b = 2 c = 3 d = 4
```

group1 is a group choice between the type choice of a and b and the type choice of c and d. This becomes more relevant once member keys and/or occurrences are added in:

```
t = {group2}
group2 = (? ab: a / b // cd: c / d)
a = 1 b = 2 c = 3 d = 4
```

is a group choice between the optional member "ab" of type a or b and the member "cd" of type c or d. Note that the optionality is attached to the first choice ("ab"), not to the second choice.

Similarly, in

```
t = [group3]
group3 = (+ a / b / c)
a = 1 b = 2 c = 3
```

group3 is a repetition of a type choice between a, b, and c [unflex]; if just a is to be repeatable, a group choice is needed to focus the occurrence:

```
t = [group4]
group4 = (+ a // b / c)
a = 1 b = 2 c = 3
```

group4 is a group choice between a repeatable a and a single b or c.

In general, as with many other languages with operator precedence rules, it is best not to rely on them, but to insert parentheses for readability:

```
t = [group4a]
group4a = ((+ a) // (b / c))
a = 1 b = 2 c = 3
```

The operator precedences, in sequence of loose to tight binding, are defined in Appendix D and summarized in Table 1. (Arities given are 1 for unary prefix operators and 2 for binary infix operators.)

Operator	Ar	Operates on	Prec
=	2	name = type, name = group	1
/=	2	name /= type	1
//=	2	name //= group	1
//	2	group // group	2
,	2	group, group	3
*	1	* group	4
N*M	1	N*M group	4
+	1	+ group	4
?	1	? group	4
=>	2	type => type	5
:	2	name: type	5
/	2	type / type	6
&	1	&group	6
..	2	type..type	7
...	2	type...type	7
.anno	2	type .anno type	7

Table 1: Summary of operator precedences

4. Making Use of CDDL

In this section, we discuss several potential ways to employ CDDL.

4.1. As a guide to a human user

CDDL can be used to efficiently define the layout of CBOR data, such that a human implementer can easily see how data is supposed to be encoded.

Since CDDL maps parts of the CBOR data to human readable names, tools could be built that use CDDL to provide a human friendly representation of the CBOR data, and allow them to edit such data while remaining compliant to its CDDL definition.

4.2. For automated checking of CBOR data structure

CDDL has been specified such that a machine can handle the CDDL definition and related CBOR data (and, thus, also JSON data). For example, a machine could use CDDL to check whether or not CBOR data is compliant to its definition.

The need for thoroughness of such compliance checking depends on the application. For example, an application may decide not to check the data structure at all, and use the CDDL definition solely as a means to indicate the structure of the data to the programmer.

On the other end, the application may also implement a checking mechanism that goes as far as checking that all mandatory map members are available.

The matter in how far the data description must be enforced by an application is left to the designers and implementers of that application, keeping in mind related security considerations.

In no case the intention is that a CDDL tool would be "writing code" for an implementation.

4.3. For data analysis tools

In the long run, it can be expected that more and more data will be stored using the CBOR data format.

Where there is data, there is data analysis and the need to process such data automatically. CDDL can be used for such automated data processing, allowing tools to verify data, clean it, and extract particular parts of interest from it.

Since CBOR is designed with constrained devices in mind, a likely use of it would be small sensors. An interesting use would thus be automated analysis of sensor data.

5. Security considerations

This document presents a content rules language for expressing CBOR data structures. As such, it does not bring any security issues on itself, although specification of protocols that use CBOR naturally need security analysis when defined.

Topics that could be considered in a security considerations section that uses CDDL to define CBOR structures include the following:

- o Where could the language maybe cause confusion in a way that will enable security issues?

6. IANA considerations

This document does not require any IANA registrations.

7. Acknowledgements

CDDL was originally conceived by Bert Greevenbosch, who also wrote the original five versions of this document.

Inspiration was taken from the C and Pascal languages, MPEG's conventions for describing structures in the ISO base media file format, Relax-NG and its compact syntax [RELAXNG], and in particular from Andrew Lee Newton's "JSON Content Rules" [I-D.newton-json-content-rules].

Useful feedback came from Joe Hildebrand, Sean Leonard and Jim Schaad.

The CDDL tool was written by Carsten Bormann, building on previous work by Troy Heninger and Tom Lord.

8. References

8.1. 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>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<http://www.rfc-editor.org/info/rfc3629>>.
- [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>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<http://www.rfc-editor.org/info/rfc7049>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<http://www.rfc-editor.org/info/rfc7159>>.
- [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<http://www.rfc-editor.org/info/rfc7493>>.

8.2. Informative References

- [I-D.ietf-anima-grasp]
Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-14 (work in progress), July 2017.
- [I-D.ietf-core-senml]
Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Media Types for Sensor Measurement Lists (SenML)", draft-ietf-core-senml-10 (work in progress), July 2017.
- [I-D.ietf-cose-msg]
Schaad, J., "CBOR Object Signing and Encryption (COSE)", draft-ietf-cose-msg-24 (work in progress), November 2016.
- [I-D.newton-json-content-rules]
Newton, A. and P. Cordell, "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-08 (work in progress), March 2017.
- [RELAXNG] OASIS, "RELAX-NG Compact Syntax", November 2002, <<http://relaxng.org/compact-20021121.html>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<http://www.rfc-editor.org/info/rfc4648>>.
- [RFC7071] Borenstein, N. and M. Kucherawy, "A Media Type for Reputation Interchange", RFC 7071, DOI 10.17487/RFC7071, November 2013, <<http://www.rfc-editor.org/info/rfc7071>>.
- [RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <<http://www.rfc-editor.org/info/rfc8007>>.

Appendix A. Cemetery

The following ideas have been buried in the discussions leading up to the present specification:

- o <...> as syntax for enumerations. We view values to be just another type (a very specific type with just one member), so that an enumeration can be denoted as a choice using "/" as the delimiter of choices. Because of this, no evidence is present that a separate syntax for enumerations is needed.

A.1. Resolved Issues

- o The key/value pairs in maps have no fixed ordering. One could imagine situations where fixing the ordering may be of use. For example, a decoder could look for values related with integer keys 1, 3 and 7. If the order were fixed and the decoder encounters the key 4 without having encountered key 3, it could conclude that key 3 is not available without doing more complicated bookkeeping. Unfortunately, neither JSON nor CBOR support this, so no attempt was made to support this in CDDL either.
- o CDDL distinguishes the various CBOR number types, but there is only one number type in JSON. There is no effect in specifying a precision (float16/float32/float64) when using CDDL for specifying JSON data structures. (The current validator implementation Appendix F does not handle this very well, either.)

Appendix B. (Not used.)

Appendix C. Change Log

Changes from version 00 to version 01:

- o Removed constants
- o Updated the tag mechanism
- o Extended the map structure
- o Added examples

Changes from version 01 to version 02:

- o Fixed example

Changes from version 02 to version 03:

- o Added information about characters used in names
- o Added text about an overall data structure and order of definition of fields
- o Added text about encoding of keys
- o Added table with keywords
- o Strings and integer writing conventions

- o Added ABNF

Changes from version 03 to version 04:

- o Removed optional fields for non-maps
- o Defined all key/value pairs in maps are considered optional from the CDDL perspective
- o Allow omission of type of keys for maps with only text string and integer keys
- o Changed order of definitions
- o Updated fruit and moves examples
- o Renamed the "Philosophy" section to "Using CDDL", and added more text about CDDL usage
- o Several editorials

Changes from version 04 to version 05:

- o Added text about alternative datatypes and any datatype
- o Fixed typos
- o Restructured syntax and semantics

Changes from version 05 to version 05:

- o Fixed the ABNF for choices (no longer need to write a: (b/c))
- o Added group choices (//)
- o Added /= and //=
- o Added experimental socket/plug
- o Added aliases text, bytes, null to prelude
- o Documented generics
- o Fixed more typos

Changes from 06 to 07:

- o .cbor, .cborseq, .within, .and

- o Define `.size` on `uint`
- o Extended Diagnostic Notation
- o Precedence discussion and table
- o Remove some of the "issues" that can only be understood with historical context
- o Prefer `"text"` over `"tstr"` in some of the examples
- o Add `"unsigned"` to the prelude

Changes from 07 to 08:

- o `.lt`, `.le`, `.eq`, `.ne`, `.gt`, `.ge`
- o `.default`

Changes from 08 to 09:

- o Take annotations and socket/plug out of the nursery; they have been battle-proven enough.
- o Define a value notation for byte strings as well.
- o Removed discussion section that was no longer relevant; move "Resolved Issues" to appendix.

Changes from 09 to 10:

- o Remove a long but not very elucidating example. (Maybe we'll add back some shorter examples later.)
- o A few clarifications.
- o Updated author list.

Changes from 10 to 11:

- o Define unwrapping operator `~`
- o Change term for annotation into `"control"` (but leave `"annotate"` for when it actually is meant in that sense)

Appendix D. ABNF grammar

The following is a formal definition of the CDDL syntax in Augmented Backus-Naur Form (ABNF, [RFC5234]). [_abnftodo]

```

cddl = S 1*rule
rule = typename [genericparm] S assign S type S
      / groupname [genericparm] S assign S grpent S

typename = id
groupname = id

assign = "=" / "/=" / "//="

genericparm = "<" S id S *(", " S id S ) ">"
genericarg = "<" S type1 S *(", " S type1 S ) ">"

type = type1 S *("/" S type1 S)

type1 = type2 [S (rangeop / ctlop) S type2]

type2 = value
      / typename [genericarg]
      / "(" type ")"
      / "~" S groupname [genericarg]
      / "#" "6" [ "." uint ] "(" S type S )" " ; note no space!
      / "#" DIGIT [ "." uint ] ; major/ai
      / "#" ; any
      / "{" S group S "}"
      / "[" S group S "]"
      / "&" S "(" S group S )" "
      / "&" S groupname [genericarg]

rangeop = "... " / ".. "

ctlop = "." id

group = grpchoice S *("//" S grpchoice S)

grpchoice = *grpent

grpent = [occur S] [memberkey S] type optcom
      / [occur S] groupname [genericarg] optcom ; preempted by above
      / [occur S] "(" S group S )" " optcom

memberkey = type1 S "=>"
          / bareword S ":"
          / value S ":"

```

```

bareword = id

optcom = S ["," S]

occur = [uint] "*" [uint]
        / "+"
        / "?"

uint = ["0x" / "0b"] "0"
        / DIGIT1 *DIGIT
        / "0x" 1*HEXDIG
        / "0b" 1*BINDIG

value = number
        / text
        / bytes

int = ["-"] uint

; This is a float if it has fraction or exponent; int otherwise
number = int ["." fraction] ["e" exponent ]
fraction = 1*DIGIT
exponent = int

text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x23-5B / %x5D-10FFFD / SESC
SESC = "\" %x20-10FFFD

bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
bsqual = %x68 ; "h"
        / %x62.36.34 ; "b64"

id = EALPHA *(("-" / ".") (EALPHA / DIGIT))
ALPHA = %x41-5A / %x61-7A
EALPHA = %x41-5A / %x61-7A / "@" / "_" / "$"
DIGIT = %x30-39
DIGIT1 = %x31-39
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
BINDIG = %x30-31

S = *WS
WS = SP / NL
SP = %x20
NL = COMMENT / CRLF
COMMENT = ";" *PCHAR CRLF
PCHAR = %x20-10FFFD
CRLF = %x0A / %x0D.0A

```

Figure 11: CDDL ABNF

Appendix E. Standard Prelude

The following prelude is automatically added to each CDDL file [tdate]. (Note that technically, it is a postlude, as it does not disturb the selection of the first rule as the root of the definition.)

```
any = #

uint = #0
nint = #1
int = uint / nint

bstr = #2
bytes = bstr
tstr = #3
text = tstr

tdate = #6.0(tstr)
time = #6.1(number)
number = int / float
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
decfrac = #6.4([e10: int, m: integer])
bigfloat = #6.5([e2: int, m: integer])
eb64url = #6.21(any)
eb64legacy = #6.22(any)
eb16 = #6.23(any)
encoded-cbor = #6.24(bstr)
uri = #6.32(tstr)
b64url = #6.33(tstr)
b64legacy = #6.34(tstr)
regexp = #6.35(tstr)
mime-message = #6.36(tstr)
cbor-any = #6.55799(any)

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
undefined = #7.23
```

Figure 12: CDDL Prelude

Note that the prelude is deemed to be fixed. This means, for instance, that additional tags beyond [RFC7049], as registered, need to be defined in each CDDL file that is using them.

A common stumbling point is that the prelude does not define a type "string". CBOR has byte strings ("bytes" in the prelude) and text strings ("text"), so a type that is simply called "string" would be ambiguous.

E.1. Use with JSON

The JSON generic data model (implicit in [RFC7159]) is a subset of the generic data model of CBOR. So one can use CDDL with JSON by limiting oneself to what can be represented in JSON. Roughly speaking, this means leaving out byte strings, tags, and simple values other than "false", "true", and "null", leading to the following limited prelude:

```
any = #

uint = #0
nint = #1
int = uint / nint

tstr = #3
text = tstr

number = int / float

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
```

Figure 13: JSON compatible subset of CDDL Prelude

(The major types given here do not have a direct meaning in JSON, but they can be interpreted as CBOR major types translated through Section 4 of [RFC7049].)

There are a few fine points in using CDDL with JSON. First, JSON does not distinguish between integers and floating point numbers; there is only one kind of number (which may happen to be integral). In this context, specifying a type as "uint", "nint" or "int" then becomes a predicate that the number be integral. As an example, this means that the following JSON numbers are all matching "uint":

```
10 10.0 1e1 1.0e1 100e-1
```

(The fact that these are all integers may be surprising to users accustomed to the long tradition in programming languages of using decimal points or exponents in a number to indicate a floating point literal.)

Fundamentally, the number system of JSON itself is based on decimal numbers and decimal fractions and does not have limits to its precision or range. In practice, JSON numbers are often parsed into a number type that is called float64 here, creating a number of limitations to the generic data model [RFC7493]. In particular, this means that integers can only be expressed with interoperable exactness when they lie in the range $[-(2^{53})+1, (2^{53})-1]$ -- a smaller range than that covered by CDDL "int".

JSON applications that want to stay compatible with I-JSON therefore may want to define integer types with more limited ranges, such as in Figure 14. Note that the types given here are not part of the prelude; they need to be copied into the CDDL specification if needed.

```
ij-uint = 0..9007199254740991
ij-nint = -9007199254740991..-1
ij-int = -9007199254740991..9007199254740991
```

Figure 14: I-JSON types for CDDL (not part of prelude)

JSON applications that do not need to stay compatible with I-JSON and that actually may need to go beyond the 64-bit unsigned and negative integers supported by "int" (= "uint"/"nint") may want to use the following additional types from the standard prelude, which are expressed in terms of tags but can straightforwardly be mapped into JSON (but not I-JSON) numbers:

```
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
```


CDDL at this point does not have a way to express the unlimited floating point precision that is theoretically possible with JSON; at the time of writing, this is rarely used in protocols in practice.

Note that a data model described in CDDL is always restricted by what can be expressed in the serialization; e.g., floating point values such as NaN (not a number) and the infinities cannot be represented in JSON even if they are allowed in the CDDL generic data model.

Appendix F. The CDDL tool

A rough CDDL tool is available. For CDDL specifications, it can check the syntax, generate one or more instances (expressed in CBOR diagnostic notation or in pretty-printed JSON), and validate an existing instance against the specification:

```
Usage:
cddl spec.cddl generate [n]
cddl spec.cddl json-generate [n]
cddl spec.cddl validate instance.cbor
cddl spec.cddl validate instance.json
```

Figure 15: CDDL tool usage

Install on a system with a modern Ruby via:

```
gem install cddl
```

Figure 16: CDDL tool installation

The accompanying CBOR diagnostic tools (which are automatically installed by the above) are described in <https://github.com/cabo/cbor-diag>; they can be used to convert between binary CBOR, a pretty-printed form of that, CBOR diagnostic notation, JSON, and YAML.

Appendix G. Extended Diagnostic Notation

Section 6 of [RFC7049] defines a "diagnostic notation" in order to be able to converse about CBOR data items without having to resort to binary data. Diagnostic notation is based on JSON, with extensions for representing CBOR constructs such as binary data and tags.

(Standardizing this together with the actual interchange format does not serve to create another interchange format, but enables the use of a shared diagnostic notation in tools for and documents about CBOR.)

This section discusses a few extensions to the diagnostic notation that have turned out to be useful since RFC 7049 was written. We refer to the result as extended diagnostic notation (EDN).

G.1. White space in byte string notation

Examples often benefit from some white space (spaces, line breaks) in byte strings. In extended diagnostic notation, white space is ignored in prefixed byte strings; for instance, the following are equivalent:

```
h'48656c6c6f20776f726c64'
h'48 65 6c 6c 6f 20 77 6f 72 6c 64'
h'4 86 56c 6c6f
 20776 f726c64'
```

G.2. Text in byte string notation

Diagnostic notation notates Byte strings in one of the [RFC4648] base encodings,, enclosed in single quotes, prefixed by >h< for base16, >b32< for base32, >h32< for base32hex, >b64< for base64 or base64url. Quite often, byte strings carry bytes that are meaningfully interpreted as UTF-8 text. Extended Diagnostic Notation allows the use of single quotes without a prefix to express byte strings with UTF-8 text; for instance, the following are equivalent:

```
'hello world'
h'68656c6c6f20776f726c64'
```

The escaping rules of JSON strings are applied equivalently for text-based byte strings, e.g., \ stands for a single backslash and ' stands for a single quote. White space is included literally, i.e., the previous section does not apply to text-based byte strings.

G.3. Embedded CBOR and CBOR sequences in byte strings

Where a byte string is to carry an embedded CBOR-encoded item, or more generally a sequence of zero or more such items, the diagnostic notation for these zero or more CBOR data items, separated by commata, can be enclosed in << and >> to notate the byte string resulting from encoding the data items and concatenating the result. For instance, each pair of columns in the following are equivalent:

```
<<1>>          h'01'
<<1, 2>>       h'0102'
<<"foo", null>> h'636666F6FF6'
<<>>           h''
```

G.4. Concatenated Strings

While the ability to include white space enables line-breaking of encoded byte strings, a mechanism is needed to be able to include text strings as well as byte strings in direct UTF-8 representation into line-based documents (such as RFCs and source code).

We extend the diagnostic notation by allowing multiple text strings or multiple byte strings to be notated separated by white space, these are then concatenated into a single text or byte string, respectively. Text strings and byte strings do not mix within such a concatenation, except that byte string notation can be used inside a sequence of concatenated text string notation to encode characters that may be better represented in an encoded way. The following four values are equivalent:

```
"Hello world"
"Hello " "world"
"Hello" h'20' "world"
"" h'48656c6c6f20776f726c64' ""
```

Similarly, the following byte string values are equivalent

```
'Hello world'
'Hello ' 'world'
'Hello ' h'776f726c64'
'Hello' h'20' 'world'
'' h'48656c6c6f20776f726c64' '' b64''
h'4 86 56c 6c6f' h' 20776 f726c64'
```

(Note that the approach of separating by whitespace, while familiar from the C language, requires some attention - a single comma makes a big difference here.)

G.5. Hexadecimal, octal, and binary numbers

In addition to JSON's decimal numbers, EDN provides hexadecimal, octal and binary numbers in the usual C-language notation (octal with 0o prefix present only).

The following are equivalent:

```
4711
0x1267
0o11147
0b1001001100111
```

As are:

```
1.5
0x1.8p0
0x18p-4
```

G.6. Comments

Longer pieces of diagnostic notation may benefit from comments. JSON famously does not provide for comments, and basic RFC 7049 diagnostic notation inherits this property.

In extended diagnostic notation, comments can be included, delimited by slashes ("/"). Any text within and including a pair of slashes is considered a comment.

Comments are considered white space. Hence, they are allowed in prefixed byte strings; for instance, the following are equivalent:

```
h'68656c6c6f20776f726c64'
h'68 65 6c /doubled l!/ 6c 6f /hello/
  20 /space/
  77 6f 72 6c 64' /world/
```

This can be used to annotate a CBOR structure as in:

```
/grasp-message/ [/M_DISCOVERY/ 1, /session-id/ 10584416,
                  /objective/ [/objective-name/ "opsonize",
                              /D, N, S/ 7, /loop-count/ 105]]
```

(There are currently no end-of-line comments. If we want to add them, "//" sounds like a reasonable delimiter given that we already use slashes for comments, but we also could go e.g. for "#".)

Appendix H. Examples

This section contains various examples of structures defined using CDDL.

The theme for the first example is taken from [RFC7071], which defines certain JSON structures in English. For a similar example, it may also be of interest to examine Appendix A of [RFC8007], which contains a CDDL definition for a JSON structure defined in the main body of the RFC.

The second subsection in this appendix translates examples from [I-D.newton-json-content-rules] into CDDL.

These examples all happen to describe data that is interchanged in JSON. Examples for CDDL definitions of data that is interchanged in

CBOR can be found in [I-D.ietf-cose-msg], [I-D.ietf-anima-grasp], or [I-D.ietf-core-senml].

H.1. RFC 7071

[RFC7071] defines the Reputon structure for JSON using somewhat formalized English text. Here is a (somewhat verbose) equivalent definition using the same terms, but notated in CDDL:

```
reputation-object = {  
    reputation-context,  
    reputon-list  
}  
  
reputation-context = (  
    application: text  
)  
  
reputon-list = (  
    reputons: reputon-array  
)  
  
reputon-array = [* reputon]  
  
reputon = {  
    rater-value,  
    assertion-value,  
    rated-value,  
    rating-value,  
    ? conf-value,  
    ? normal-value,  
    ? sample-value,  
    ? gen-value,  
    ? expire-value,  
    * ext-value,  
}  
  
rater-value = ( rater: text )  
assertion-value = ( assertion: text )  
rated-value = ( rated: text )  
rating-value = ( rating: float16 )  
conf-value = ( confidence: float16 )  
normal-value = ( normal-rating: float16 )  
sample-value = ( sample-size: uint )  
gen-value = ( generated: uint )  
expire-value = ( expires: uint )  
ext-value = ( text => any )
```

An equivalent, more compact form of this example would be:

```
reputation-object = {  
  application: text  
  reputons: [* reputon]  
}  
  
reputon = {  
  rater: text  
  assertion: text  
  rated: text  
  rating: float16  
  ? confidence: float16  
  ? normal-rating: float16  
  ? sample-size: uint  
  ? generated: uint  
  ? expires: uint  
  * text => any  
}
```

Note how this rather clearly delineates the structure somewhat shrouded by so many words in section 6.2.2. of [RFC7071]. Also, this definition makes it clear that several ext-values are allowed (by definition with different member names); RFC 7071 could be read to forbid the repetition of ext-value ("A specific reputon-element MUST NOT appear more than once" is ambiguous.)

The CDDL tool (which hasn't quite been trained for polite conversation) says:

```
{
  "application": "tridentiferous",
  "reputons": [
    {
      "rater": "loamily",
      "assertion": "Dasypsecta",
      "rated": "uncommensurableness",
      "rating": 0.05055809746548934,
      "confidence": 0.7484706448605812,
      "normal-rating": 0.8677887734049299,
      "sample-size": 4059,
      "expires": 3969,
      "bearer": "nitty",
      "faucal": "postulnar",
      "naturalism": "sarcotic"
    },
    {
      "rater": "precreed",
      "assertion": "xanthosis",
      "rated": "balsamy",
      "rating": 0.36091333590593955,
      "confidence": 0.3700759808403371,
      "sample-size": 3904
    },
    {
      "rater": "urinosexual",
      "assertion": "malacostracous",
      "rated": "arenariae",
      "rating": 0.9210673488013762,
      "normal-rating": 0.4778762617112776,
      "sample-size": 4428,
      "generated": 3294,
      "backfurrow": "enterable",
      "fruitgrower": "flannelflower"
    },
    {
      "rater": "pedologistically",
      "assertion": "unmetaphysical",
      "rated": "elocutionist",
      "rating": 0.42073613384304287,
      "misimagine": "retinaculum",
      "snobbish": "contradict",
      "Bosporanic": "periostotomy",
      "dayworker": "intragyrar"
    }
  ]
}
```

H.1.1.1. Examples from JSON Content Rules

Although JSON Content Rules [I-D.newton-json-content-rules] seems to address a more general problem than CDDL, it is still a worthwhile resource to explore for examples (beyond all the inspiration the format itself has had for CDDL).

Figure 2 of the JCR I-D looks very similar, if slightly less noisy, in CDDL:

```
root = [2*2 {  
    precision: text,  
    Latitude: float,  
    Longitude: float,  
    Address: text,  
    City: text,  
    State: text,  
    Zip: text,  
    Country: text  
}]
```

Figure 17: JCR, Figure 2, in CDDL

Apart from the lack of a need to quote the member names, text strings are called "text" or "tstr" in CDDL ("string" would be ambiguous as CBOR also provides byte strings).

The CDDL tool creates the below example instance for this:

```
[{"precision": "pyrosphere", "Latitude": 0.5399712314350172,  
  "Longitude": 0.5157523963028087, "Address": "resow",  
  "City": "problemwise", "State": "martyrlike", "Zip": "preprove",  
  "Country": "Pace"},  
{"precision": "unrigging", "Latitude": 0.10422704368372193,  
  "Longitude": 0.6279808663725834, "Address": "picturedom",  
  "City": "decipherability", "State": "autometry", "Zip": "pout",  
  "Country": "wimple"}]
```

Figure 4 of the JCR I-D in CDDL:


```
root = { image }

image = (
  Image: {
    size,
    Title: text,
    thumbnail,
    IDs: [* int]
  }
)

size = (
  Width: 0..1280
  Height: 0..1024
)

thumbnail = (
  Thumbnail: {
    size,
    Url: ~uri
  }
)
```

This shows how the group concept can be used to keep related elements (here: width, height) together, and to emulate the JCR style of specification. (It also shows referencing a type by unwrapping a tag from the prelude, "uri" - this could be done differently.) The more compact form of Figure 5 of the JCR I-D could be emulated like this:

```
root = {
  Image: {
    size, Title: text,
    Thumbnail: { size, Url: ~uri },
    IDs: [* int]
  }
}

size = (
  Width: 0..1280,
  Height: 0..1024,
)
```

The CDDL tool creates the below example instance for this:

```
{"Image": {"Width": 566, "Height": 516, "Title": "leisterer",
  "Thumbnail": {"Width": 1111, "Height": 176, "Url": 32("scrog")},
  "IDs": []}}
```

Editorial Comments

- [`_format`] So far, the ability to restrict format choices have not been needed beyond the floating point formats. Those can be applied to ranges using the new `.and` control now. It is not clear we want to add more format control before we have a use case.
- [`_range`] TO DO: define this precisely. This clearly includes integers and floats. Strings – as in `"a".."z"` – could be added if desired, but this would require adopting a definition of string ordering and possibly a successor function so `"a".."z"` does not include `"bb"`.
- [`_strings`] TO DO: This still needs to be fully realized in the ABNF and in the CDDL tool.
- [`_bitendian`] How useful would it be to have another variant that counts bits like in RFC box notation? (Or at least per-byte? 32-bit words don't always perfectly mesh with byte strings.)
- [`unflex`] A comment has been that this is counter-intuitive. One solution would be to simply disallow unparenthesized usage of occurrence indicators in front of type choices unless a member key is also present like in `group2` above.
- [`_abnftodo`] Potential improvements: the prefixed byte strings are more liberally specified than they actually are. [`^_abnfdontdo`]: representation indicators are not supported. – and this will stay so.
- [`tdate`] The prelude as included here does not yet have a `.regex` control on `tdate`, but we probably do want to have one.

Authors' Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de

Christoph Vigano
Universitaet Bremen

Email: christoph.vigano@uni-bremen.de

Carsten Bormann
Universitaet Bremen TZI
Bibliothekstr. 1
Bremen D-28359
Germany

Phone: +49-421-218-63921
Email: cabo@tzi.org

APPSAWG
Internet-Draft
Obsoletes: 2388 (if approved)
Intended status: Standards Track
Expires: October 12, 2015

L. Masinter
Adobe
April 10, 2015

Returning Values from Forms: multipart/form-data
draft-ietf-appsawg-multipart-form-data-11

Abstract

This specification defines the multipart/form-data Internet Media Type, which can be used by a wide variety of applications and transported by a wide variety of protocols as a way of returning a set of values as the result of a user filling out a form. It obsoletes RFC 2388.

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 October 12, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. percent-encoding option	3
3. Advice for Forms and Form Processing	3
4. Definition of multipart/form-data	4
4.1. Boundary parameter of multipart/form-data	4
4.2. Content-Disposition header for each part	4
4.3. filename attribute of content-distribution part header	4
4.4. Multiple files for one form field	5
4.5. Content-Type header for each part	5
4.6. The charset parameter for text/plain form data	5
4.7. The _charset_ field for default charset	6
4.8. Content-Transfer-Encoding deprecated	6
4.9. Other Content- headers	7
5. Operability considerations	7
5.1. Non-ASCII field names and values	7
5.1.1. Avoid non-ASCII field names	7
5.1.2. Interpreting forms and creating form-data	7
5.1.3. Parsing and interpreting form data	8
5.2. Ordered fields and duplicated field names	8
5.3. Interoperability with web applications	8
5.4. Correlating form data with the original form	9
6. IANA Considerations	9
7. Security Considerations	9
8. Media type registration for multipart/form-data	10
9. References	11
9.1. Normative References	11
9.2. Informative References	12
Appendix A. Changes from RFC 2388	12
Appendix B. Alternatives	13
Author's Address	13

1. Introduction

In many applications, it is possible for a user to be presented with a form. The user will fill out the form, including information that is typed, generated by user input, or included from files that the user has selected. When the form is filled out, the data from the form is sent from the user to the receiving application.

The definition of "multipart/form-data" is derived from one of those applications, originally set out in [RFC1867] and subsequently incorporated into HTML 3.2 [W3C.REC-html32-19970114], where forms are expressed in HTML, and in which the form data is sent via HTTP or

electronic mail. This representation is widely implemented in numerous web browsers and web servers.

However, "multipart/form-data" is also used for forms that are presented using representations other than HTML (spreadsheets, PDF, etc.), and for transport using means other than electronic mail or HTTP; it is used in distributed applications which do not involve forms at all, or do not have users filling out the form. For this reason, this document defines a general syntax and semantics independent of the application for which it is used, with specific rules for web applications noted in context.

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

2. percent-encoding option

Within this specification, "percent-encoding" (as defined in [RFC3986]) is offered as a possible way of encoding characters in file names that are otherwise disallowed, including non-ASCII characters, spaces, control characters and so forth. The encoding is created replacing each non-ASCII or disallowed character with a sequence, where each byte of the UTF-8 encoding of the character is represented by a percent-sign (%) followed by the (case-insensitive) hexadecimal of that byte.

3. Advice for Forms and Form Processing

The representation and interpretation of forms and the nature of form processing is not specified by this document. However, for forms and form-processing that result in generation of multipart/form-data, some suggestions are included.

In a form, there is generally a sequence of fields, where each field is expected to be supplied with a value, e.g. by a user who fills out the form. Each field has a name. After a form has been filled out, and the form's data is "submitted": the form processing results in a set of values for each field-- the "form data".

In forms that work with multipart/form-data, field names could be arbitrary Unicode strings; however, restricting field names to ASCII will help avoid some interoperability issues (see Section 5.1).

Within a given form, ensuring field names are unique is also helpful. Some fields may have default values or presupplied values in the form itself. Fields with presupplied values might be hidden or invisible;

this allows using generic processing for form data from a variety of actual forms.

4. Definition of multipart/form-data

The media-type "multipart/form-data" follows the model of multipart MIME data streams as specified in [RFC2046] Section 5.1; changes are noted in this document.

A "multipart/form-data" body contains a series of parts, separated by a boundary.

4.1. Boundary parameter of multipart/form-data

As with other multipart types, the parts are delimited with a boundary delimiter, constructed using CRLF, "--", the value of the boundary parameter. The boundary is supplied as a "boundary" parameter to the "multipart/form-data" type. As noted in [RFC2046] Section 5.1, the boundary delimiter MUST NOT appear inside any of the encapsulated parts, and it is often necessary to enclose the boundary parameter values in quotes on the Content-type line.

4.2. Content-Disposition header for each part

Each part MUST contain a "content-disposition" header [RFC2183] and where the disposition type is "form-data". The "content-disposition" header MUST also contain an additional parameter of "name"; the value of the "name" parameter is the original field name from the form (possibly encoded; see Section 5.1). For example, a part might contain a header:

```
Content-Disposition: form-data; name="user"
```

with the body of the part containing the form data of the "user" field.

4.3. filename attribute of content-distribution part header

For form data that represents the content of a file, a name for the file SHOULD be supplied as well, by using a "filename" parameter of the "content-disposition" header. The file name isn't mandatory for cases where the file name isn't available or is meaningless or private; this might result, for example, from selection or drag-and-drop or where the form data content is streamed directly from a device.

If a filename parameter is supplied, the requirements of [RFC2183] Section 2.3 for "receiving MUA" apply to receivers of "multipart/

form-data" as well: Do not use the file name blindly, check and possibly change to match local filesystem conventions if applicable, do not use directory path information that may be present.

In most multipart types, the MIME headers in each part are restricted to US-ASCII; for compatibility with those systems, file names normally visible to users MAY be encoded using the percent-encoding method in Section 2, following how a "file:" URI [I-D.ietf-appsawg-file-scheme] might be encoded.

NOTE: The encoding method described in [RFC5987], which would add a "filename*" paramter to the "Content-Disposition" header, MUST NOT be used.

Some commonly deployed systems use multipart/form-data with file names directly encoded including octets outside the US-ASCII range. The encoding used for the file names is typically UTF-8, although HTML forms will use the charset associated with the form.

4.4. Multiple files for one form field

The form data for a form field might include multiple files.

[RFC2388] suggested that multiple files for a single form field be transmitted using a nested multipart/mixed part. This usage is deprecated.

To match widely deployed implementations, multiple files MUST be sent by supplying each file in a separate part, but all with the same "name" parameter.

Receiving applications intended for wide applicability (e.g. multipart/form-data parsing libraries) SHOULD also support the older method of supplying multiple files.

4.5. Content-Type header for each part

Each part MAY have an (optional) "content-type", which defaults to "text/plain". If the contents of a file are to be sent, the file data SHOULD be labeled with an appropriate media type, if known, or "application/octet-stream".

4.6. The charset parameter for text/plain form data

In the case where the form data is text, the charset parameter for the "text/plain" Content-Type MAY be used to indicate the character encoding used in that part. For example, a form with a text field in

which a user typed "Joe owes <eu>100" where <eu> is the Euro symbol might have form data returned as:

```
--AaB03x
content-disposition: form-data; name="field1"
content-type: text/plain;charset=UTF-8
content-transfer-encoding: quoted-printable

Joe owes =E2=82=AC100.
--AaB03x
```

In practice, many widely deployed implementations do not supply a charset parameter in each part, but, rather, they rely on the notion of a "default charset" for a multipart/form-data instance. Subsequent sections will explain how the default charset is established.

4.7. The `_charset_` field for default charset

Some form processing applications (including HTML) have the convention that the value of a form entry with entry name "`_charset_`" and type "hidden" is automatically set when the form is opened; the value is used as the default charset of text field values (see form-charset in Section 5.1.2). In such cases, the value of the default charset for each text/plain part without a charset parameter is the supplied value. For example:

```
--AaB03x
content-disposition: form-data; name="_charset_"

iso-8859-1
--AaB03x--
content-disposition: form-data; name="field1"

...text encoded in iso-8859-1 ...
AaB03x--
```

4.8. Content-Transfer-Encoding deprecated

Previously, it was recommended that senders use a "Content-Transfer-Encoding" encoding (such as "quoted-printable") for each non-ASCII part of a multipart/form-data body, because that would allow use in transports that only support a "7BIT" encoding. This use is deprecated for use in contexts that support binary data such as HTTP. Senders SHOULD NOT generate any parts with a "Content-Transfer-Encoding" header.

Currently, no deployed implementations that send such bodies have been discovered.

4.9. Other Content- headers

The "multipart/form-data" media type does not support any MIME headers in the parts other than Content-Type, Content-Disposition, and (in limited circumstances) Content-Transfer-Encoding. Other headers MUST NOT be included and MUST be ignored.

5. Operability considerations

5.1. Non-ASCII field names and values

Normally, MIME headers in multipart bodies are required to consist only of 7-bit data in the US-ASCII character set. While [RFC2388] suggested that non-ASCII field names be encoded according to the method in [RFC2047], this practice doesn't seem to have been followed widely.

This specification makes three sets of recommendations for three different states of workflow.

5.1.1. Avoid non-ASCII field names

For broadest interoperability with existing deployed software, those creating forms SHOULD avoid non-ASCII field names. This should not be a burden, because in general the field names are not visible to users. The field names in the underlying need not match what the user sees on the screen.

If non-ASCII field names are unavoidable, form or application creators SHOULD use UTF-8 uniformly. This will minimize interoperability problems.

5.1.2. Interpreting forms and creating form-data

Some applications of this specification will supply a character encoding to be used for interpretation of the multipart/form-data body. In particular, HTML 5 [W3C.REC-html5-20141028] uses:

- o The content of a '_charset_' field, if there is one.
- o the value of an accept-charset attribute of the <form> element, if there is one,
- o the character encoding of the document containing the form, if it is US-ASCII compatible,

- o otherwise UTF-8.

Call this value the form-charset. Any text, whether field name, field value, or (text/plain) form data which is uses characters outside the ASCII range MAY be represented directly encoded in the form-charset.

5.1.3. Parsing and interpreting form data

While this specification provides guidance for creation of multipart/form-data, parsers and interpreters should be aware of the variety of implementations. File systems differ as to whether and how they normalize Unicode names, for example. The matching of form elements to form-data parts may rely on a fuzzier match. In particular, some multipart/form-data generators might have followed the previous advice of [RFC2388] and used the [RFC2047] "encoded-word" method of encoding non-ASCII values:

```
encoded-word = "=?" charset "?" encoding "?" encoded-text "=?"
```

Others have been known to follow [RFC2231], to send unencoded UTF-8, or even strings encoded in the form-charset.

For this reason, interpreting "multipart/form-data" (even from conforming generators) may require knowing the charset used in form encoding, in cases where the `_charset_` field value or a charset parameter of a text/plain Content-Type header is not supplied.

5.2. Ordered fields and duplicated field names

Form processors given forms with a well-defined ordering SHOULD send back results in order (note that there are some forms which do not define a natural order.) Intermediaries MUST NOT reorder the results. Form parts with identical field names MUST NOT be coalesced.

5.3. Interoperability with web applications

Many web applications use the "application/x-url-encoded" method for returning data from forms. This format is quite compact, e.g.:

```
name=Xavier+Xantico&verdict=Yes&colour=Blue&happy=sad&Utf%F6r=Send
```

However, there is no opportunity to label the enclosed data with content type, apply a charset, or use other encoding mechanisms.

Many form-interpreting programs (primarily web browsers) now implement and generate multipart/form-data, but an existing

application might need to optionally support both the application/x-url-encoded format as well.

5.4. Correlating form data with the original form

This specification provides no specific mechanism by which multipart/form-data can be associated with the form that caused it to be transmitted. This separation is intentional; many different forms might be used for transmitting the same data. In practice, applications may supply a specific form processing resource (in HTML, the ACTION attribute in a FORM tag) for each different form. Alternatively, data about the form might be encoded in a "hidden field" (a field which is part of the form but which has a fixed value to be transmitted back to the form-data processor.)

6. IANA Considerations

Please update the Internet Media Type registration of multipart/form-data to point to this document, using the template in Section 8. In addition, please update the registrations of the "name" parameter and the "form-data" value in the "Content Disposition Values and Parameters" registry to both point to this document.

7. Security Considerations

All form processing software should treat user supplied form-data with sensitivity, as it often contains confidential or personally identifying information. There is widespread use of form "auto-fill" features in web browsers; these might be used to trick users to unknowingly send confidential information when completing otherwise innocuous tasks. Multipart/form-data does not supply any features for checking integrity, ensuring confidentiality, avoiding user confusion, or other security features; those concerns must be addressed by the form-filling and form-data-interpreting applications.

Applications which receive forms and process them must be careful not to supply data back to the requesting form processing site that was not intended to be sent.

It is important when interpreting the filename of the Content-Disposition header to not overwrite files in the recipient's file space inadvertently.

User applications that request form information from users must be careful not to cause a user to send information to the requestor or a third party unwillingly or unwittingly. For example, a form might request 'spam' information to be sent to an unintended third party,

or private information to be sent to someone that the user might not actually intend. While this is primarily an issue for the representation and interpretation of forms themselves (rather than the data representation of the form data), the transportation of private information must be done in a way that does not expose it to unwanted prying.

With the introduction of form-data that can reasonably send back the content of files from a user's file space, the possibility arises that a user might be sent an automated script that fills out a form and then sends one of the user's local files to another address. Thus, additional caution is required when executing automated scripting where form-data might include a user's files.

Files sent via multipart/form-data may contain arbitrary executable content, and precautions against malicious content are necessary.

The considerations of [RFC2183] Sections 2.3 and 5 with respect to the filename parameter of the Content-Disposition header also apply to its usage here.

8. Media type registration for multipart/form-data

This section is the [RFC6838] media type registration.

Type name: multipart

Subtype name: form-data

Required parameters: boundary

Optional parameters: none

Encoding considerations: Common use is BINARY.

In limited use (or transports that restrict the encoding to 7BIT or 8BIT each part is encoded separately using Content-Transfer-Encoding Section 4.8.

Security considerations: See Section 7 of this document.

Interoperability considerations: This document makes several recommendations for interoperability with deployed implementations, including Section 4.8.

Published specification: This document.

Applications that use this media type: Numerous web browsers, servers, and web applications.

Fragment identifier considerations: None: Fragment identifiers are not defined for this type.

Additional information: None: no deprecated alias names, magic numbers, file extensions or Macintosh sssfile type codes.

Person & email address to contact for further information
Author of this document.

Intended Usage: COMMON

Restrictions on usage: none

Author: Author of this document.

Change controller: IETF

Provisional registration: N/A

9. References

9.1. Normative References

- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.
- [RFC2047] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text", RFC 2047, November 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2183] Troost, R., Dorner, S., and K. Moore, "Communicating Presentation Information in Internet Messages: The Content-Disposition Header Field", RFC 2183, August 1997.
- [RFC2231] Freed, N. and K. Moore, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations", RFC 2231, November 1997.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

9.2. Informative References

- [I-D.ietf-appsawg-file-scheme]
Kerwin, M., "The file URI Scheme", draft-ietf-appsawg-file-scheme-00 (work in progress), January 2015.
- [RFC1867] Nebel, E. and L. Masinter, "Form-based File Upload in HTML", RFC 1867, November 1995.
- [RFC2388] Masinter, L., "Returning Values from Forms: multipart/form-data", RFC 2388, August 1998.
- [RFC5987] Reschke, J., "Character Set and Language Encoding for Hypertext Transfer Protocol (HTTP) Header Field Parameters", RFC 5987, August 2010.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [W3C.REC-html32-19970114]
Raggett, D., "HTML 3.2 Reference Specification", World Wide Web Consortium Recommendation REC-html32-19970114, January 1997, <<http://www.w3.org/TR/REC-html32-19970114>>.
- [W3C.REC-html5-20141028]
Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", World Wide Web Consortium Recommendation REC-html5-20141028, October 2014, <<http://www.w3.org/TR/2014/REC-html5-20141028>>.

Appendix A. Changes from RFC 2388

The handling of non-ASCII field names changed-- no longer recommending the RFC 2047 method, instead suggesting senders send UTF-8 field names directly, and file names directly in the form-charset.

The handling of multiple files submitted as the result of a single form field (e.g. HTML's <input type=file multiple> element) results in each file having its own top level part with the same name parameter; the method of using a nested "multipart/mixed" from [RFC2388] is no longer recommended for creators, and not required for receivers as there are no known implementations of senders.

The `_charset_` convention and use of an explicit form-data charset is documented.

'boundary' is a required parameter in Content-Type.

The relationship of the ordering of fields within a form and the ordering of returned values within multipart/form-data was not defined before, nor was the handling of the case where a form has multiple fields with the same name.

Editorial: Removed obsolete discussion of alternatives in appendix. Update references. Move outline of form processing into Introduction.

Appendix B. Alternatives

There are numerous alternative ways in which form data can be encoded; many are listed in [RFC2388] section 5.2. The multipart/form-data encoding is verbose, especially if there are many fields with short values. In most use cases, this overhead isn't significant.

More problematic are the differences introduced when implementors opted to not follow [RFC2388] when encoding non-ASCII field names (perhaps because "may" should have been "MUST"). As a result, parsers need to be more complex for matching against the possible outputs of various encoding methods.

Author's Address

Larry Masinter
Adobe

Email: masinter@adobe.com
URI: <http://larry.masinter.net>

appsawg
Internet-Draft
Updates: 3986 (if approved)
Intended status: Best Current Practice
Expires: November 22, 2014

M. Nottingham
May 21, 2014

URI Design and Ownership
draft-ietf-appsawg-uri-get-off-my-lawn-05

Abstract

RFC3986 Section 1.1.1 defines URI syntax as "a federated and extensible naming system wherein each scheme's specification may further restrict the syntax and semantics of identifiers using that scheme." In other words, the structure of a URI is defined by its scheme. While it is common for schemes to further delegate their substructure to the URI's owner, publishing independent standards that mandate particular forms of URI substructure is inappropriate, because that essentially usurps ownership. This document further describes this problematic practice and provides some acceptable alternatives for use in standards.

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 November 22, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Who This Document Is For	3
1.2. Notational Conventions	4
2. Best Current Practices for Standardizing Structured URIs . .	4
2.1. URI Schemes	4
2.2. URI Authorities	5
2.3. URI Paths	5
2.4. URI Queries	5
2.5. URI Fragment Identifiers	6
3. Alternatives to Specifying Structure in URIs	6
4. Security Considerations	7
5. IANA Considerations	7
6. References	7
6.1. Normative References	7
6.2. Informative References	8
Appendix A. Acknowledgments	8
Author's Address	8

1. Introduction

URIs [RFC3986] very often include structured application data. This might include artifacts from filesystems (often occurring in the path component), and user information (often in the query component). In some cases, there can even be application-specific data in the authority component (e.g., some applications are spread across several hostnames to enable a form of partitioning or dispatch).

Furthermore, constraints upon the structure of URIs can be imposed by an implementation; for example, many Web servers use the filename extension of the last path segment to determine the media type of the response. Likewise, pre-packaged applications often have highly structured URIs that can only be changed in limited ways (often, just the hostname and port they are deployed upon).

Because the owner of the URI (as defined in [webarch] Section 2.2.2.1) is choosing to use the server or the application, this can be seen as reasonable delegation of authority. When such conventions are mandated by a party other than the owner, however, it can have several potentially detrimental effects:

- o Collisions - As more ad hoc conventions for URI structure become standardized, it becomes more likely that there will be collisions between them (especially considering that servers, applications and individual deployments will have their own conventions).
- o Dilution - When the information added to a URI is ephemeral, this dilutes its utility by reducing its stability (see [webarch] Section 3.5.1), and can cause several alternate forms of the URI to exist (see [webarch] Section 2.3.1).
- o Rigidity - Fixed URI syntax often interferes with desired deployment patterns. For example, if an authority wishes to offer several applications on a single hostname, it becomes difficult to impossible to do if their URIs do not allow the required flexibility.
- o Operational Difficulty - Supporting some URI conventions can be difficult in some implementations. For example, specifying that a particular query parameter be used with "HTTP" URIs precludes the use of Web servers that serve the response from a filesystem. Likewise, an application that fixes a base path for its operation (e.g., "/v1") makes it impossible to deploy other applications with the same prefix on the same host.
- o Client Assumptions - When conventions are standardized, some clients will inevitably assume that the standards are in use when those conventions are seen. This can lead to interoperability problems; for example, if a specification documents that the "sig" URI query parameter indicates that its payload is a cryptographic signature for the URI, it can lead to undesirable behavior.

Publishing a standard that constrains an existing URI structure in ways which aren't explicitly allowed by [RFC3986] (usually, by updating the URI scheme definition) is inappropriate, because the structure of a URI needs to be firmly under the control of its owner, and the IETF (as well as other organizations) should not usurp it.

This document explains some best current practices for establishing URI structures, conventions and formats in standards. It also offers strategies for specifications to avoid violating these guidelines in Section 3.

1.1. Who This Document Is For

This document's requirements target the authors of specifications that constrain the syntax or structure of URIs or parts of them. Two classes of such specifications are called out specifically:

- o Protocol Extensions ("extensions") - specifications that offer new capabilities that could apply to any identifier, or to a large subset of possible identifiers; e.g., a new signature mechanism for 'http' URIs, or metadata for any URI.
- o Applications Using URIs ("applications") - specifications that use URIs to meet specific needs; e.g., a HTTP interface to particular information on a host.

Requirements that target the generic class "Specifications" apply to all specifications, including both those enumerated above and others.

Note that this specification ought not be interpreted as preventing the allocation of control of URIs by parties that legitimately own them, or have delegated that ownership; for example, a specification might legitimately define the semantics of a URI on the IANA.ORG Web site as part of the establishment of a registry.

There may be existing IETF specifications that already deviate from the guidance in this document. In these cases, it is up to the relevant communities (i.e. those of the URI scheme as well as that which produced the specification in question) to determine an appropriate outcome; e.g., updating the scheme definition, or changing the specification.

1.2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Best Current Practices for Standardizing Structured URIs

This section updates [RFC3986] by setting limitations on how other specifications may define structure and semantics within URIs. Best practices differ depending on the URI component, as described below.

2.1. URI Schemes

Applications and extensions MAY require use of specific URI scheme(s); for example, it is perfectly acceptable to require that an application support 'http' and 'https' URIs. However, applications SHOULD NOT preclude the use of other URI schemes in the future, unless they are clearly only usable with the nominated schemes.

A specification that defines substructure within a specific URI scheme **MUST** do so in the defining document for that URI scheme. A specification that defines substructure for URI schemes overall **MUST** do so by modifying [BCP115] (an exceptional circumstance).

2.2. URI Authorities

Scheme definitions define the presence, format and semantics of an authority component in URIs; all other specifications **MUST NOT** constrain, or define the structure or the semantics for URI authorities, unless they update the scheme registration itself.

For example, an extension or application ought not say that the "foo" prefix in "foo_app.example.com" is meaningful or triggers special handling in URIs.

However, applications **MAY** nominate or constrain the port they use, when applicable. For example, BarApp could run over port nnnn (provided that it is properly registered).

2.3. URI Paths

Scheme definitions define the presence, format, and semantics of a path component in URIs; all other specifications **MUST NOT** constrain, or define the structure or the semantics for any path component.

The only exception to this requirement is registered "well-known" URIs, as specified by [RFC5785]. See that document for a description of the applicability of that mechanism.

For example, an application ought not specify a fixed URI path "/"myapp", since this usurps the host's control of that space.

Specifying a fixed path relative to another (e.g., {whatever}/myapp) is also bad practice (even if "whatever" is discovered as suggested in Section 3); while doing so might prevent collisions, it does not avoid the potential for operational difficulties (for example, an implementation that prefers to use query processing instead, because of implementation constraints).

2.4. URI Queries

The presence, format and semantics of the query component of URIs is dependent upon many factors, and **MAY** be constrained by a scheme definition. Often, they are determined by the implementation of a resource itself.

Applications MUST NOT directly specify the syntax of queries, as this can cause operational difficulties for deployments that do not support a particular form of a query. For example, a site may wish to support an application using "static" files that do not support query parameters.

Extensions MUST NOT constrain the format or semantics of queries.

For example, an extension that indicates that all query parameters with the name "sig" indicate a cryptographic signature would collide with potentially pre-existing query parameters on sites, and lead clients to assume that any matching query parameter is a signature.

HTML [W3C.REC-html401-19991224] constrains the syntax of query strings used in form submission. New form languages SHOULD NOT emulate it, but instead allow creation of a broader variety of URIs (e.g., by allowing the form to create new path components, and so forth).

Note that "well-known" URIs (see [RFC5785]) MAY constrain their own query syntax, since these name spaces are effectively delegated to the registering party.

2.5. URI Fragment Identifiers

Media type definitions (as per [RFC6838]) SHOULD specify the fragment identifier syntax(es) to be used with them; other specifications MUST NOT define structure within the fragment identifier, unless they are explicitly defining one for reuse by media type definitions.

For example, an application that defines common fragment identifiers across media types not controlled by it would engender interoperability problems with handlers for those media types (because the new, non-standard syntax is not expected).

3. Alternatives to Specifying Structure in URIs

Given the issues described in Section 1, the most successful strategy for applications and extensions that wish to use URIs is to use them in the fashion they were designed: as links that are exchanged as part of the protocol, rather than statically specified syntax. Several existing specifications can aid in this.

[RFC5988] specifies relation types for Web links. By providing a framework for linking on the Web, where every link has a relation type, context and target, it allows applications to define a link's semantics and connectivity.

[RFC6570] provides a standard syntax for URI Templates that can be used to dynamically insert application-specific variables into a URI to enable such applications while avoiding impinging upon URI owners' control of them.

[RFC5785] allows specific paths to be 'reserved' for standard use on URI schemes that opt into that mechanism ('http' and 'https' by default). Note, however, that this is not a general "escape valve" for applications that need structured URIs; see that specification for more information.

Specifying more elaborate structures in an attempt to avoid collisions is not an acceptable solution, and does not address the issues in Section 1. For example, prefixing query parameters with "myapp_" does not help, because the prefix itself is subject to the risk of collision (since it is not "reserved").

4. Security Considerations

This document does not introduce new protocol artifacts with security considerations. It prohibits some practices that might lead to vulnerabilities; for example, if a security-sensitive mechanism is introduced by assuming that a URI path component or query string has a particular meaning, false positives might be encountered (due to sites that already use the chosen string). See also [RFC6943].

5. IANA Considerations

There are no direct IANA actions specified in this document.

6. References

6.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [webarch] Jacobs, I. and N. Walsh, "Architecture of the World Wide Web, Volume One", December 2004, <<http://www.w3.org/TR/2004/REC-webarch-20041215>>.

6.2. Informative References

- [BCP115] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", RFC 4395, BCP 115, February 2006, <<https://tools.ietf.org/html/bcp115>>.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.
- [RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.
- [RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, March 2012.
- [RFC6943] Thaler, D., "Issues in Identifier Comparison for Security Purposes", RFC 6943, May 2013.
- [W3C.REC-html401-19991224]
Raggett, D., Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999,
<<http://www.w3.org/TR/1999/REC-html401-19991224>>.

Appendix A. Acknowledgments

Thanks to David Booth, Dave Crocker, Tim Bray, Anne van Kesteren, Martin Thomson, Erik Wilde, Dave Thaler and Barry Leiba for their suggestions and feedback.

Author's Address

Mark Nottingham

Email: mnot@mnot.net

URI: <http://www.mnot.net/>

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 20, 2014

D. Thaler
Microsoft
October 17, 2013

Guidelines and Registration Procedures for New URI Schemes: Problem
Statement
draft-thaler-uri-scheme-reg-ps-01.txt

Abstract

This document describes some problems with the existing guidelines and procedures, as documented in RFC 4395, for new URI schemes.

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 20, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Problems	4
2.1. Current registration process doesn't scale well	4
2.2. Lack of incentive to register	5
2.3. Current private scheme guidance causes conflicts	5
3. Security Considerations	6
4. IANA Considerations	6
5. Informative References	6
Author's Address	7

1. Introduction

RFC 4395 [RFC4395] provides guidelines and recommendations for the definition of Uniform Resource Identifier (URI) schemes. It defines procedures and guidelines for four types of URI schemes:

- a. Permanent, which [RFC4395] requires for all IETF Standards-Track schemes, and which has strict requirements.
- b. Provisional, which has a lower barrier.
- c. Historical, which is for schemes no longer in use and hence generally does not apply to "new" URI schemes.
- d. Private, meaning not registered with IANA.

As explained in Section 1 of [RFC4395], the purpose of an IANA-maintained registry is to:

1. provide a central point of discovery for established URI scheme names, and easy location of their defining documents;
2. discourage use of the same URI scheme name for different purposes;
3. help those proposing new URI scheme names to discern established trends and conventions, and avoid names that might be confused with existing ones;
4. encourage registration by setting a low barrier for provisional registrations.

However, the guidance in [RFC4395] is, in many cases that are now common, ambiguous or insufficient to accomplish the stated purposes. This document discusses a number of such problems. In doing so, we note that an effort was started to update the guidance, in

[I-D.ietf-iri-4395bis-irireg]. It does not, however, address the problems we discuss in this document, although it may be the logical place to do so.

It is first important to understand the scale of the problem. It is already common on many widely deployed platforms (including Windows, iOS, and Android) and form factors (PCs, phones, etc.) today to allow applications to be associated with specific URI schemes, such that when the URI is accessed (e.g., clicking on a link in a browser, or calling an equivalent API from an application), the associated application is launched to handle the URI. That is, the application is given the URI and determines what action to take (as opposed to being given content that the URI points to). Indeed, some such URIs are simply Uniform Resource Names that contain the data themselves, rather than Uniform Resource Locators that can be resolved to content. As such, URIs are increasingly becoming a form of inter-process communication as a way to invoke another application, with arguments placed in the scheme-specific part of the URI. Thus, in the extreme case, every application might define its own URI scheme, and the number of applications available on mainstream platforms today is easily numbered in the hundreds of thousands.

This use of URIs can be viewed as different from the web. That is, an increasingly larger portion of URI schemes are intended for "local" use, rather than for use with the web. The "URI Generic Syntax" [RFC3986] explicitly allows for such a wide scope of use of URIs. It states, in section 1.1:

This specification does not limit the scope of what might be a resource; rather, the term "resource" is used in a general sense for whatever might be identified by a URI. Familiar examples include an electronic document, an image, a source of information with a consistent purpose (e.g., "today's weather report for Los Angeles"), a service (e.g., an HTTP-to-SMS gateway), and a collection of other resources. A resource is not necessarily accessible via the Internet; e.g., human beings, corporations, and bound books in a library can also be resources. Likewise, abstract concepts can be resources, such as the operators and operands of a mathematical equation, the types of a relationship (e.g., "parent" or "employee"), or numeric values (e.g., zero, one, and infinity).

and

This specification does not place any limits on the nature of a resource, the reasons why an application might seek to refer to a resource, or the kinds of systems that might use URIs for the sake of identifying resources.

The current process was designed based in part on joint recommendations from the W3C and IETF in 2002 [RFC3305], when the known uses of schemes were such that there were 34 registered schemes, 51 known publically documented but unregistered schemes, and 50 or so private schemes with 2-3 being added every day, as noted (see Section 3.1 of [RFC3305]). Such private growth has continued and expanded to more platforms since then, such that the public schemes are now probably a small minority.

2. Problems

2.1. Current registration process doesn't scale well

Section 5.2 of [RFC4395] requires a four-week mailing list review for all Permanent registrations. It is, however, ambiguous as to whether a mailing list review is required for Provisional registrations and if so, for how long. The longer the process, the less of an incentive there is to register Provisional schemes. This problem was discussed in 2010 by the IRI WG, which concluded that a mailing list review should not be required for Provisional schemes, only expert review which may take up to two weeks, but this conclusion has not yet been documented.

The manual step of expert review still introduces a scalability bottleneck. What if all new applications being submitted to an app store started sending requests for Provisional URI schemes? The expert review process would be overwhelmed, especially if no one is paid to do the expert review. As such, the goals stated in Section 1 become far less effective when registered schemes are only a tiny fraction of the URI schemes in use in practice.

The author ran an experiment in 2012, which was reported to the IRI WG at its final meeting at IETF 85, where over 75 schemes that were listed on Wikipedia as being unregistered but in use were submitted as third-party registrations. All of them were registered after two weeks had passed and it was pointed out that the deadline had expired and per the process in [RFC4395], must be automatically listed. The only noticeable outcome of the expert review, other than to introduce a two week delay and manual effort, was to add a warning about the unknown security impact of one scheme. This is not intended to imply that the expert review was not valuable, only that the value provided could not scale effectively if the process were stressed with the current potential demand.

In summary, [RFC4395] defines a set of goals, which we listed above in Section 1. The current mechanism does not meet those goals. To meet the stated goals would require the majority of schemes to be registered. The current process cannot scale to do so, given current

practice. Hence, we either need to change the goals, or change the process, or both.

2.2. Lack of incentive to register

Currently there is little incentive for an organization outside the IETF to register schemes (whether as Permanent, Provisional, or Historical). Registering introduces a cost, both in terms of manual effort needed to apply, but also in the time delay introduced. This cost must be weighed against the benefit, which is primarily to simply lower the risk of collision. (Another benefit is to provide ease of access to relevant documentation via the IANA registry, although this benefit is often seen as unimportant or even undesirable in some cases.)

As long as the risk of collision is perceived to be low, or the effect of collision considered to be acceptable (e.g., asking the user which app to launch), registration is bypassed in favor of a "Private" scheme. The effect of collision can of course be problematic (though the scheme-defining organization may not realize the danger) when the syntax of the scheme-specific part differs. Launching an application with a URI that is invalid according to that application's syntax for the custom URI scheme is not useful.

An app store certification process could in theory require or encourage Provisional application. However, there is little incentive for them to do so either, since an app store itself has a process which would be delayed and disincent application developers to submit applications.

2.3. Current private scheme guidance causes conflicts

Section 2.8 of [RFC4395] states:

Organizations that desire a private name space for URI scheme names are encouraged to use a prefix based on their domain name, expressed in reverse order. For example, a URI scheme name of com-example-info might be registered by the vendor that owns the example.com domain name.

There are multiple problems with the above guidance:

1. No guidance is given for when it might or might not be appropriate to use a private name space. For example, is this guidance appropriate for application vendors defining a custom scheme that they want to associate the application with? As such, the current assumption is that it is appropriate for anyone who can live with some potential risk of collision.

2. Hyphens occur in actual domain names. Consider one organization that owns the domain name "foo.bar.example", and another organization that owns "foo-bar.example". Using the mechanism implied in the example can result in both colliding with "example-bar-foo-info".
3. The guidance is only an encouragement, and no precise algorithm is given. For example, whether "." should be converted to "-" as in the example is unclear. If an organization is actually trying to follow the recommended guidelines, they will likely use a "-" as directed and risk conflicts as noted above. More commonly, an organization today will simply use a string that identifies (say) their application, and not be based on a domain name.
4. No protection is suggested against IANA later granting registration to a scheme that follows the recommended convention that is in use by someone else. For example, as can be seen at [IANAURI], there are already registered schemes that use "." (e.g., "iris.beep") and "-" (e.g., "xcon-userid") in them, and there could be similar new schemes registered at any time. If an organization had previously acquired the TLD "iris" or "xcon", those values could already be in use in applications from those organizations. Especially now that ICANN is allowing gTLD applications, this is a very real possibility.

3. Security Considerations

The security considerations in [RFC4395] still apply.

4. IANA Considerations

This document requires no actions by the IANA.

5. Informative References

[I-D.ietf-iri-4395bis-irireg]

Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI/IRI Schemes", draft-ietf-iri-4395bis-irireg-04 (work in progress), December 2011.

[IANAURI]

IANA, ., "Uniform Resource Identifier (URI) Schemes", 2013, <<http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml>>.

[RFC3305]

Mealling, M. and R. Denenberg, "Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names

(URNs): Clarifications and Recommendations", RFC 3305, August 2002.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4395] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 35, RFC 4395, February 2006.

Author's Address

Dave Thaler
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA

Phone: +1 425 703 8835
Email: dthaler@microsoft.com

IETF
Internet-Draft
Intended status: Standards Track
Expires: October 18, 2014

N. Tomkinson
N. Borenstein
Mimecast Ltd
Apr 16, 2014

Multiple Language Content Type
draft-tomkinson-multilangcontent-01

Abstract

This document defines an addition to the Multipurpose Internet Mail Extensions (MIME) standard to make it possible to send one message that contains multiple language versions of the same information. The translations would be identified by a language code and selected by the email client based on a user's language settings or locale.

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 October 18, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

Since the invention of email and the rapid spread of the internet, more and more people have been able to communicate in more and more countries and in more and more languages. But during this time of technological evolution, email has remained a single language communication tool, whether it is English to English, Spanish to Spanish or Japanese to Japanese.

Also during this time, many corporations have established their offices in multi-cultural cities and formed departments and teams that span continents, cultures and languages so the need to communicate efficiently with little margin for miscommunication has grown exponentially.

The objective of this document is to define an addition to the widely used Multipurpose Internet Mail Extensions (MIME) standard, to make it possible to send a single message to a group of people in such a way that all of the recipients can read the email in their own first language. The methods of translation of the message content are beyond the scope of this document, but the structure of the email itself is presented herein.

1.1. Requirements Language

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

2. The Content-Type Header

When there is a requirement to send a message in a number of different languages and the translations are to be embedded in the same message, the multipart subtype "multipart/multilingual" SHOULD be used to help the receiving email client make sense of the message structure.

The suggested multipart subtype "multipart/multilingual" has the same semantics as "multipart/alternative" (as discussed in RFC 2046 [RFC2046]) in that each of the body parts is an alternative version of the same information. The primary difference between "multipart/multilingual" and "multipart/alternative" is that when using "multipart/multilingual", the message part to select for rendering is chosen based on the value of the Content-Language header instead of the ordering of the parts and the Content-Types.

The syntax for this multipart subtype conforms to the common syntax

for subtypes of multipart given in section 5.1.1. of RFC 2046 [RFC2046], therefore, an example "multipart/multilingual" Content-Type header field would look like this:

```
Content-type: multipart/multilingual; boundary=01189998819991197253
```

3. The Multilingual Preface -- the first message part

In order for the message to be received and displayed in non-conforming email clients, the message MUST contain an explanatory message part which MUST-NOT be marked with a Content-Language header but MUST be the first of the message parts. Because non-conforming email clients are expected to treat the message as multipart/mixed (in accordance with sections 5.1.3 and 5.1.7 of RFC 2046 [RFC2046]) they may show all of the message parts sequentially or as attachments. Including and showing this explanatory part will help the message recipient understand the message structure.

This initial message part SHOULD explain briefly to the message recipient that the message contains multiple languages and the parts may be rendered sequentially or as attachments. This SHOULD be presented in the same languages that are provided in the subsequent message parts.

Whilst this section of the message is useful for backward compatibility, it SHOULD only be shown when rendered by a non-conforming email client because conforming email client SHOULD only show the single message part identified by the user's preferred language (or locale) and the message part's Content-Language.

4. The Subsequent Message Parts

The subsequent message parts are translations of the same message content. These body parts MAY be ordered so that the first part after the multilingual preface is in the language believed to be the most likely to be recognised by recipients using software that does not implement multipart/multilingual.

The Content-Type for each individual language part MAY be any MIME type (including multipart subtypes such as multipart/alternative). However, it is recommended that the Content-Type of the language parts is kept as simple as possible for interoperability with existing email clients. The language parts are not required to have matching Content-Types or multipart structures. For example, there might be an English part of type "text/html" followed by a Spanish part of type "application/pdf" followed by a Chinese part of type

"image/jpeg". Whatever the content-type, the contents SHOULD be composed for optimal viewing in the specified language.

5. The Content-Language Header

The Content-Language header in the individual multipart message parts is used to identify the language in which the message part is written. Based on the value of this header, a conforming email client can determine which message part to display (given the user's language settings or locale).

The Content-Language MUST comply with RFC 3282 [RFC3282] (which defines the Content-Language header) and BCP 47/RFC 5646 [RFC5646] (which defines the structure and semantics for the language code values). Examples of this header for English, English as used in the United States and Latin American Spanish, could look like the following:

Content-Language: en

Content-Language: en-US

Content-Language: es-419

6. The Subject-Translation Header

On receipt of the message, conforming email clients will need to select the correct multipart message content and replace the subject that is shown to the message recipient with the translated subject. To enable this the Subject-Translation header SHOULD be provided in each message part that contains a Content-Language header.

The value for this header should be a simple translated version of the original email subject. An example of this header may look like this:

Subject-Translation: Mensaje de ejemplo para varios idiomas

7. Examples

7.1. An Example of a Simple Multiple language email message

Below is an example of a simple multiple language email message formatted using the method detailed in this document.

From: Nik
To: Nathaniel
Subject: Simple example multiple language message
Content-type: multipart/multilingual; boundary=01189998819991197253

--01189998819991197253

This is a message in two languages: English and Spanish. It says the same thing in each language. If you read it in the first language, you can ignore the other translations. The other translations may be presented as attachments or grouped together.

Este es un mensaje en dos idiomas: Ingles y Espanol. Dice lo mismo en cada idioma. Si lo necesita en el primer idioma, puede ignorar el otras traducciones. Las otras traducciones se pueden presentar como archivos adjuntos o agrupados.

--01189998819991197253

Content-Language: en
Content-Type: text/plain
Subject-Translation: Simple example multiple language message

Hello, this message content is provided in your language.

--01189998819991197253

Content-Language: es
Content-Type: text/plain
Subject-Translation: Ejemplo simple mensaje en varios idiomas

Hola, el contenido de este mensaje esta disponible en su idioma.

--01189998819991197253--

7.2. An Example of a Complex Multiple language email message

Below is an example of a more complex multiple language email message formatted using the method detailed in this document. Note that the language parts have multipart contents and would therefore require further processing to determine the content to display.

From: Nik
To: Nathaniel
Subject: Complex example multiple language message
Content-type: multipart/multilingual; boundary=01189998819991197253

--01189998819991197253

This is a message in two languages: English and Spanish. It says the same thing in each language. If you read it in the first language, you can ignore the other translations. The other translations may be presented as attachments or grouped together.

Este es un mensaje en dos idiomas: Ingles y Espanol. Dice lo mismo en cada idioma. Si lo necesita en el primer idioma, puede ignorar el otras traducciones. Las otras traducciones se pueden presentar como archivos adjuntos o agrupados.

--01189998819991197253

Content-Language: en

Content-Type: multipart/alternative; boundary=multipartaltboundary

Subject-Translation: Complex example multiple language message

--multipartaltboundary

Content-Type: text/plain

Hello, this message content is provided in your language.

--multipartaltboundary

Content-Type: text/html

<html><body><p>Hello, this message content is provided in your language.<p></body></html>

--multipartaltboundary--

--01189998819991197253

Content-Language: es

Content-Type: multipart/mixed; boundary=multipartmixboundary

Subject-Translation: Ejemplo complejo mensaje en varios idiomas

--multipartmixboundary

Content-Type: application/pdf

..PDF file in Spanish here..

--multipartmixboundary

Content-Type: image/jpeg

..JPEG image showing Spanish content here..

--multipartmixboundary--

--01189998819991197253--

8. Acknowledgements

The authors are grateful for the helpful input received from many people but would especially like to acknowledge the help of Harald Alvestrand, Mark Davis, Doug Ewell, Fiona Tomkinson and Simon Tyler.

9. IANA Considerations

The multipart/multilingual MIME type will be registered with IANA.

10. Security Considerations

This document has no additional security considerations beyond those that apply to the standards and procedures on which it is built.

11. Normative References

- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3282] Alvestrand, H., "Content Language Headers", RFC 3282, May 2002.
- [RFC5646] Phillips, A. and M. Davis, "Tags for Identifying Languages", BCP 47, RFC 5646, September 2009.

Authors' Addresses

Nik Tomkinson
Mimecast Ltd
CityPoint, One Ropemaker Street
London, EC2Y 9AW
United Kingdom

Email: rfc.nik.tomkinson@gmail.com

Nathaniel Borenstein
Mimecast Ltd
480 Pleasant Street
Watertown, MA 02472
North America

Email: nsb@mimecast.com

