Binary Encodings for JavaScript Object Notation: JSON-B, JSON-C, JSON-D
                      draft-hallambaker-jsonbcd-01

Abstract

   Three binary encodings for JavaScript Object Notation (JSON) are
   presented. JSON-B (Binary) is a strict superset of the JSON encoding
   that permits efficient binary encoding of intrinsic JavaScript data
   types. JSON-C (Compact) is a strict superset of JSON-B that supports
   compact representation of repeated data strings with short numeric
   codes. JSON-D (Data) supports additional binary data types for
   integer and floating point representations for use in scientific
   applications where conversion between binary and decimal
   representations would cause a loss of precision.

Status of This Memo

Copyright Notice

Table of Contents

1. Definitions

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 [RFC2119].

2. Introduction

JavaScript Object Notation (JSON) is a simple text encoding for the
JavaScript Data model that has found wide application beyond its
original field of use. In particular JSON has rapidly become a
preferred encoding for Web Services.

JSON encoding supports just four fundamental data types (integer,
floating point, string and boolean), arrays and objects which consist
of a list of tag-value pairs.

Although the JSON encoding is sufficient for many purposes it is not
always efficient. In particular there is no efficient representation
for blocks of binary data. Use of base64 encoding increases data
volume by 33%. This overhead increases exponentially in applications
where nested binary encodings are required making use of JSON

encoding unsatisfactory in cryptographic applications where nested
binary structures are frequently required.

Another source of inefficiency in JSON encoding is the repeated
occurrence of object tags. A JSON encoding containing an array of a
hundred objects such as {"first":1,"second":2} will contain a hundred
occurrences of the string "first" (seven bytes) and a hundred
occurrences of the string "second" (eight bytes). Using two byte
code sequences in place of strings allows a saving of 11 bytes per
object without loss of information, a saving of 50%.

A third objection to the use of JSON encoding is that floating point
numbers can only be represented in decimal form and this necessarily
involves a loss of precision when converting between binary and
decimal representations. While such issues are rarely important in
network applications they can be critical in scientific applications.
It is not acceptable for saving and restoring a data set to change
the result of a calculation.

2.1. Objectives

The following were identified as core objectives for a binary JSON
encoding:

   Low overhead encoding and decoding

   Easy to convert existing encoders and decoders to add binary
   support

   Efficient encoding of binary data

   Ability to convert from JSON to binary encoding in a streaming
   mode (i.e. without reading the entire binary data block before
   beginning encoding.

   Lossless encoding of JavaScript data types

   The ability to support JSON tag compression and extended data
   types are considered desirable but not essential for typical
   network applications.

Three binary encodings are defined:

JSON-B (Binary) Simply encodes JSON data in binary. Only the
   JavaScript data model is supported (i.e. atomic types are
   integers, double or string). Integers may be 8, 16, 32 or 64 bits
   either signed or unsigned. Floating points are IEEE 754 binary64

format [IEEE-754]. Supports chunked encoding for binary and UTF-8 string types.

JSON-C (Compact) As JSON-B but with support for representing JSON tags in numeric code form (16 bit code space). This is done for both compact encoding and to allow simplification of encoders/ decoders in constrained environments. Codes may be defined inline or by reference to a known dictionary of codes referenced via a digest value.

JSON-D (Data) As JSON-C but with support for representing additional data types without loss of precision. In particular other IEEE 754 floating point formats, both binary and decimal and Intel's 80 bit floating point, plus 128 bit integers and bignum integers.

3. Extended JSON Grammar

The JSON-B, JSON-C and JSON-D encodings are all based on the JSON grammar [RFC4627] using the same syntactic structure but different lexical encodings.

JSON-B0 and JSON-C0 replace the JSON lexical encodings for strings and numbers with binary encodings. JSON-B1 and JSON-C1 allow either lexical encoding to be used. Thus any valid JSON encoding is a valid JSON-B1 or JSON-C1 encoding.

The grammar of JSON-B, JSON-C and JSON-D is a superset of the JSON grammar. The following productions are added to the grammar:

x-value Binary encodings for data values. As the binary value encodings are all self delimiting

x-member An object member where the value is specified as an X-value and thus does not require a value-separator.

b-value Binary data encodings defined in JSON-B.

b-string Defined length string encoding defined in JSON-B.

c-def Tag code definition defined in JSON-C. These may only appear before the beginning of an Object or Array and before any preceeding white space.

c-tag Tag code value defined in JSON-C.

d-value Additional binary data encodings defined in JSON-D for use in scientific data applications.

The JSON grammar is modified to permit the use of x-value productions
in place of (value value-separator) :

JSON-text = (object / array)

    object = *cdef begin-object [
        *(member value-separator | x-member)
          (member | x-member)] end-object

    member = tag value
    x-member = tag x-value

    tag = string name-separator | b-string | c-tag

    array = *cdef begin-array [*(value value-separator | x-value)
        (value | x-value)] end-array

    x-value = b-value / d-value

    value = false / null / true / object / array / number / string

    name-separator = ws %x3A ws ; : colon
    value-separator = ws %x2C ws ; , comma

The following lexical values are unchanged:

    begin-array   = ws %x5B ws ; [left square bracket
    begin-object   = ws %x7B ws ; { left curly bracket
    end-array   = ws %x5D ws ;] right square bracket
    end-object   = ws %x7D ws ; } right curly bracket

    ws = *(%x20 %x09 %x0A %x0D)

    false = %x66.61.6c.73.65   ; false
    null = %x6e.75.6c.6c   ; null
    true = %x74.72.75.65   ; true

The productions number and string are defined as before:

```
      number = [ minus ] int [ frac ] [ exp ]
      decimal-point = %x2E        ; .
      digit1-9 = %x31-39          ; 1-9
      e = %x65 / %x45             ; e E
      exp = e [ minus / plus ] 1*DIGIT
      frac = decimal-point 1*DIGIT
      int = zero / ( digit1-9 *DIGIT )
      minus = %x2D                ; -
      plus = %x2B                 ; +
      zero = %x30                 ; 0

      string = quotation-mark *char quotation-mark
      char = unescaped /
             escape ( %x22 / %x5C / %x2F / %x62 / %x66 /
                      %x6E / %x72 / %x74 /  %x75 4HEXDIG )

      escape = %x5C               ; \
      quotation-mark = %x22       ; "
      unescaped = %x20-21 / %x23-5B / %x5D-10FFFF
```

4. JSON-B

   The JSON-B encoding defines the b-value and b-string productions:

```
      b-value = b-atom | b-string | b-data | b-integer |
            b-float

      b-string = *( string-chunk ) string-term
      b-data = *( data-chunk ) data-last

      b-integer = p-int8 | p-int16 | p-int32 | p-int64 | p-bignum16 |
                  n-int8 | n-int16 | n-int32 | n-int64 | n-bignum16

      b-float = binary64
```

   The lexical encodings of the productions are defined in the following
   table where the column 'tag' specifies the byte code that begins the
   production, 'Fixed' specifies the number of data bytes that follow
   and 'Length' specifies the number of bytes used to define the length
   of a variable length field following the data bytes:

Production	Tag	Fixed	Length	Data Description
string-term	x80	–	1	Terminal String 8 bit length
string-term	x81	–	2	Terminal String 16 bit length

Name	Code			Description
string-term	x82	-	4	Terminal String 32 bit length
string-term	x83	-	8	Terminal String 64 bit length
string-chunk	x84	-	1	Non-Terminal String 8 bit length
string-chunk	x85	-	2	Non-Terminal String 16 bit length
string-chunk	x86	-	4	Non-Terminal String 32 bit length
string-chunk	x87	-	8	Non-Terminal String 64 bit length
data-term	x88	-	1	Terminal Data 8 bit length
data-term	x89	-	2	Terminal Data 16 bit length
data-term	x8A	-	4	Terminal Data 32 bit length
data-term	x8B	-	8	Terminal Data 64 bit length
data-chunk	x8C	-	1	Non-Terminal Data 8 bit length
data-chunk	x8D	-	2	Non-Terminal Data 16 bit length
data-chunk	x8E	-	4	Non-Terminal Data 32 bit length
data-chunk	x8F	-	8	Non-Terminal String 64 bit length
p-int8	xA0	1	-	Positive 8 bit Integer
p-int16	xA1	2	-	Positive 16 bit Integer
p-int32	xA2	4	-	Positive 32 bit Integer
p-int64	xA3	8	-	Positive 64 bit Integer
p-bignum16	xA5	-	2	Positive Bignum 16 bit length
n-int8	xA8	1	-	Negative 8 bit Integer
n-int16	xA9	2	-	Negative 16 bit Integer
n-int32	xAA	4	-	Negative 32 bit Integer
n-int64	xAB	8	-	Negative 64 bit Integer
n-bignum16	xAD	-	2	Negative Bignum 16 bit length
binary64	x92	8	-	IEEE 754 Floating Point binary64
b-value	xB0	-	-	True
b-value	xB1	-	-	False
b-value	xB2	-	-	Null

Table 1: JSON-B Lexical Encodings

A data type commonly used in networking that is not defined in this scheme is a datetime representation.

4.1. JSON-B Examples

   The following examples show examples of using JSON-B encoding:

               Binary Encoding   JSON Equivalent

   A0 2A   42 (as 8 bit integer)
   A1 00 2A   42 (as 16 bit integer)
   A2 00 00 00 2A   42 (as 32 bit integer)
   A3 00 00 00 00 00 00 00 2A   42 (as 64 bit integer)
   A5 00 01 42   42 (as Bignum)

   80 05 48 65 6c 6c 6f   "Hello" (single chunk)
   81 00 05 48 65 6c 6c 6f   "Hello" (single chunk)
   84 05 48 65 6c 6c 6f 80 00   "Hello" (as two chunks)

   92 3f f0 00 00 00 00 00 00   1.0
   92 40 24 00 00 00 00 00 00   10.0
   92 40 09 21 fb 54 44 2e ea   3.14159265359
   92 bf f0 00 00 00 00 00 00   -1.0

   B0   true
   B1   false
   B2   null

5. JSON-C

   JSON-C (Compressed) permits numeric code values to be substituted for
   strings and binary data. Tag codes MAY be 8, 16 or 32 bits long
   encoded in network byte order.

   Tag codes MUST be defined before they are referenced. A Tag code MAY
   be defined before the corresponding data or string value is used or
   at the same time that it is used.

   A dictionary is a list of tag code definitions. An encoding MAY
   incorporate definitions from a dictionary using the dict-hash
   production. The dict hash production specifies a (positive) offset
   value to be added to the entries in the dictionary and a hash code
   identifier consisting of the ASN.1 OID value sequence for the
   cryptographic digest used to compute the hash value followed by the
   hash value in network byte order.

```
+------------+-----+-------+--------+------------------------------+
| Production | Tag | Fixed | Length |       Data Description       |
+------------+-----+-------+--------+------------------------------+
|    c-tag   | xC0 |   1   |   -    |       8 bit tag code         |
|    c-tag   | xC1 |   2   |   -    |      16 bit tag code         |
|    c-tag   | xC2 |   4   |   -    |      32 bit tag code         |
|    c-def   | xC4 |   1   |   -    |    8 bit tag definition       |
|    c-def   | xC5 |   2   |   -    |   16 bit tag definition       |
|    c-def   | xC6 |   4   |   -    |   32 bit tag definition       |
|    c-tag   | xC8 |   1   |   -    | 8 bit tag code & definition   |
|    c-tag   | xC9 |   2   |   -    | 16 bit tag code & definition  |
|    c-tag   | xCA |   4   |   -    | 32 bit tag code & definition  |
|    c-def   | xCC |   1   |   -    |     8 bit tag dictionary      |
|            |     |       |        |         definition           |
|    c-tag   | xCD |   2   |   -    |    16 bit tag dictionary      |
|            |     |       |        |         definition           |
|    c-tag   | xCE |   4   |   -    |    32 bit tag dictionary      |
|            |     |       |        |         definition           |
| dict-hash  | xD0 |   4   |   1    |     Hash of dictionary        |
+------------+-----+-------+--------+------------------------------+
```

                    Table 2: JSON-C Lexical Encodings

   All integer values are encoded in Network Byte Order (most
   significant byte first).

5.1. JSON-C Examples

   The following examples show examples of using JSON-C encoding:

```
   JSON-C                            Value      Define

   C8 20 80 05 48 65 6c 6c 6f        "Hello"    20 = "Hello"
   C4 21 80 05 48 65 6c 6c 6f                   21 = "Hello"
   C0 20                             "Hello"
   C1 00 20                          "Hello"

   D0 00 00 01 00 1B                            277 = "Hello"
      06 09 60 86 48 01 65 03
      04 02 01                       OID for SHA-2-256
      e3 b0 c4 42 98 fc 1c 14
      9a fb f4 c8 99 6f b9 24
      27 ae 41 e4 64 9b 93 4c
      a4 95 99 1b 78 52 b8 55        SHA-256(C4 21 80 05 48 65 6c 6c 6f)



   2.16.840.1.101.3.4.2.1
```

6. **JSON-D (Data)**

   JSON-B and JSON-C only support the two numeric types defined in the
   JavaScript data model: Integers and 64 bit floating point values.
   JSON-D (Data) defines binary encodings for additional data types that
   are commonly used in scientific applications. These comprise
   positive and negative 128 bit integers, six additional floating point
   representations defined by IEEE 754 [RFC2119] and the Intel extended
   precision 80 bit floating point representation.

   Should the need arise, even bigger bignums could be defined with the
   length specified as a 32 bit value permitting bignums of up to 2^35
   bits to be represented.

```
                 d-value = d-integer | d-float

       d-float = binary16 | binary32 | binary128 | binary80 |
                 decimal32 | decimal64 | decimal 128
```

Production	Tag	Fixed	Length	Data Description
p-int128	xA4	16	-	Positive 128 bit Integer
n-in7128	xAC	16	-	Negative 128 bit Integer
binary16	x90	2	-	IEEE 754 Floating Point binary16
binary32	x91	4	-	IEEE 754 Floating Point binary32
binary128	x94	16	-	IEEE 754 Floating Point binary128
intel80	x95	10	-	Intel 80 bit extended binary Floating Point
decimal32	x96	4	-	IEEE 754 Floating Point decimal32
decimal64	x97	8	-	IEEE 754 Floating Point decimal64
decimal128	x98	18	-	IEEE 754 Floating Point decimal128

                   Table 3: JSON-D Lexical Encodings

7. Acknowledgements

   Nico Williams, etc

8. Security Considerations

9. IANA Considerations

   [TBS list out all the code points that require an IANA registration]

10. Normative References

   [IEEE-754]
               "Information technology -- Microprocessor Systems --
               Floating-Point arithmetic", ISO/IEC/IEEE 60559:2011, July
               2011, <IEEE-754>.

   [RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate
               Requirement Levels", BCP 14, RFC 2119, March 1997.

   [RFC4627]   Crockford, D., "The application/json Media Type for
               JavaScript Object Notation (JSON)", RFC 4627, July 2006.

Author's Address

   Phillip Hallam-Baker
   Comodo Group Inc.

   Email: philliph@comodo.com