Network Working Group
Request for Comments: 4627
Category: Informational
D. Crockford
JSON.org
July 2006

The application/json Media Type for JavaScript Object Notation (JSON)

Status of This Memo

Copyright Notice

Abstract

JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data.

1. Introduction

JavaScript Object Notation (JSON) is a text format for the serialization of structured data. It is derived from the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard, Third Edition [ECMA].

JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).

A string is a sequence of zero or more Unicode characters [UNICODE].

An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.

An array is an ordered sequence of zero or more values.

The terms "object" and "array" come from the conventions of JavaScript.

JSON's design goals were for it to be minimal, portable, textual, and a subset of JavaScript.

1.1. Conventions Used in This Document

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].

The grammatical rules in this document are to be interpreted as described in [RFC4234].

2. JSON Grammar

A JSON text is a sequence of tokens. The set of tokens includes six structural characters, strings, numbers, and three literal names.

A JSON text is a serialized object or array.

```
JSON-text = object / array
```

These are the six structural characters:

```
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

name-separator  = ws %x3A ws  ; : colon

value-separator = ws %x2C ws  ; , comma
```

Insignificant whitespace is allowed before or after any of the six structural characters.

```
ws = *(
          %x20 /              ; Space
          %x09 /              ; Horizontal tab
          %x0A /              ; Line feed or New line
          %x0D                ; Carriage return
      )
```

2.1. Values

A JSON value MUST be an object, array, number, or string, or one of the following three literal names:

```
false null true
```

The literal names MUST be lowercase. No other literal names are allowed.

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

false = %x66.61.6c.73.65   ; false

null  = %x6e.75.6c.6c      ; null

true  = %x74.72.75.65      ; true
```

2.2. Objects

An object structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members). A name is a string. A single colon comes after each name, separating the name from the value. A single comma separates a value from a following name. The names within an object SHOULD be unique.

```
object = begin-object [ member *( value-separator member ) ]
end-object

member = string name-separator value
```

2.3. Arrays

An array structure is represented as square brackets surrounding zero or more values (or elements). Elements are separated by commas.

```
array = begin-array [ value *( value-separator value ) ] end-array
```

2.4. Numbers

The representation of numbers is similar to that used in most programming languages. A number contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part.

Octal and hex forms are not allowed. Leading zeros are not allowed.

A fraction part is a decimal point followed by one or more digits.

An exponent part begins with the letter E in upper or lowercase, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits.

Numeric values that cannot be represented as sequences of digits (such as Infinity and NaN) are not permitted.

```
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
```

2.5. Strings

The representation of strings is similar to conventions used in the C family of programming languages. A string begins and ends with quotation marks. All Unicode characters may be placed within the quotation marks except for the characters that must be escaped: quotation mark, reverse solidus, and the control characters (U+0000 through U+001F).

Any character may be escaped. If the character is in the Basic Multilingual Plane (U+0000 through U+FFFF), then it may be represented as a six-character sequence: a reverse solidus, followed by the lowercase letter u, followed by four hexadecimal digits that encode the character's code point. The hexadecimal letters A though F can be upper or lowercase. So, for example, a string containing only a single reverse solidus character may be represented as "\u005C".

Alternatively, there are two-character sequence escape representations of some popular characters. So, for example, a string containing only a single reverse solidus character may be represented more compactly as "\\".

To escape an extended character that is not in the Basic Multilingual Plane, the character is represented as a twelve-character sequence, encoding the UTF-16 surrogate pair. So, for example, a string containing only the G clef character (U+1D11E) may be represented as "\uD834\uDD1E".

```
string = quotation-mark *char quotation-mark

char = unescaped /
       escape (
           %x22 /          ; "    quotation mark  U+0022
           %x5C /          ; \    reverse solidus U+005C
           %x2F /          ; /    solidus         U+002F
           %x62 /          ; b    backspace       U+0008
           %x66 /          ; f    form feed       U+000C
           %x6E /          ; n    line feed       U+000A
           %x72 /          ; r    carriage return U+000D
           %x74 /          ; t    tab             U+0009
           %x75 4HEXDIG )  ; uXXXX                U+XXXX

escape = %x5C              ; \

quotation-mark = %x22      ; "

unescaped = %x20-21 / %x23-5B / %x5D-10FFFF
```

3. Encoding

JSON text SHALL be encoded in Unicode. The default encoding is UTF-8.

Since the first two characters of a JSON text will always be ASCII characters [RFC0020], it is possible to determine whether an octet stream is UTF-8, UTF-16 (BE or LE), or UTF-32 (BE or LE) by looking at the pattern of nulls in the first four octets.

```
00 00 00 xx  UTF-32BE
00 xx 00 xx  UTF-16BE
xx 00 00 00  UTF-32LE
xx 00 xx 00  UTF-16LE
xx xx xx xx  UTF-8
```

4. Parsers

A JSON parser transforms a JSON text into another representation. A JSON parser MUST accept all texts that conform to the JSON grammar. A JSON parser MAY accept non-JSON forms or extensions.

An implementation may set limits on the size of texts that it accepts. An implementation may set limits on the maximum depth of nesting. An implementation may set limits on the range of numbers. An implementation may set limits on the length and character contents of strings.

5. Generators

A JSON generator produces JSON text. The resulting text MUST strictly conform to the JSON grammar.

6. IANA Considerations

The MIME media type for JSON text is application/json.

Type name: application

Subtype name: json

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: 8bit if UTF-8; binary if UTF-16 or UTF-32

JSON may be represented using UTF-8, UTF-16, or UTF-32. When JSON is written in UTF-8, JSON is 8bit compatible. When JSON is written in UTF-16 or UTF-32, the binary content-transfer-encoding must be used.

Security considerations:

Generally there are security issues with scripting languages. JSON is a subset of JavaScript, but it is a safe subset that excludes assignment and invocation.

A JSON text can be safely passed into JavaScript's eval() function (which compiles and executes a string) if all the characters not enclosed in strings are in the set of characters that form JSON tokens. This can be quickly determined in JavaScript with two regular expressions and calls to the test and replace methods.

```
var my_JSON_object = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test(
       text.replace(/"(\\.|[^"\\])*"/g, ''))) &&
   eval('(' + text + ')');
```

Interoperability considerations: n/a

Published specification: RFC 4627

Applications that use this media type:

JSON has been used to exchange data between applications written in all of these programming languages: ActionScript, C, C#, ColdFusion, Common Lisp, E, Erlang, Java, JavaScript, Lua, Objective CAML, Perl, PHP, Python, Rebol, Ruby, and Scheme.

Additional information:

Magic number(s): n/a
File extension(s): .json
Macintosh file type code(s): TEXT

Person & email address to contact for further information:
Douglas Crockford
douglas@crockford.com

Intended usage: COMMON

Restrictions on usage: none

Author:
Douglas Crockford
douglas@crockford.com

Change controller:
Douglas Crockford
douglas@crockford.com

7. Security Considerations

See Security Considerations in Section 6.

8. Examples

This is a JSON object:

```
{
   "Image": {
       "Width":  800,
       "Height": 600,
       "Title":  "View from 15th Floor",
       "Thumbnail": {
           "Url":    "http://www.example.com/image/481989943",
           "Height": 125,
           "Width":  "100"
       },
       "IDs": [116, 943, 234, 38793]
```

```
      }
   }
```

Its Image member is an object whose Thumbnail member is an object and whose IDs member is an array of numbers.

This is a JSON array containing two objects:

```
   [
      {
         "precision": "zip",
         "Latitude":  37.7668,
         "Longitude": -122.3959,
         "Address":   "",
         "City":      "SAN FRANCISCO",
         "State":     "CA",
         "Zip":       "94107",
         "Country":   "US"
      },
      {
         "precision": "zip",
         "Latitude":  37.371991,
         "Longitude": -122.026020,
         "Address":   "",
         "City":      "SUNNYVALE",
         "State":     "CA",
         "Zip":       "94085",
         "Country":   "US"
      }
   ]
```

9. References

9.1. Normative References

[ECMA] European Computer Manufacturers Association, "ECMAScript Language Specification 3rd Edition", December 1999, <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>.

[RFC0020] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.

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

[RFC4234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005.

[UNICODE] The Unicode Consortium, "The Unicode Standard Version 4.0", 2003, <http://www.unicode.org/versions/Unicode4.1.0/>.

Author's Address

Douglas Crockford
JSON.org
EMail: douglas@crockford.com

Acknowledgement

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).