Operations Area Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 08, 2013

D. Crockford
JSON.org
June 06, 2013

The JSON Data Interchange Format
draft-ietf-json-rfc4627bis-01

Abstract

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.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

JSON is a text format for the serialization of structured data. It was inspired by the object literals of JavaScript, as defined in the ECMAScript Programming Language Standard, Fifth 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 characters.

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. JSON stands for JavaScript Object Notation.

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 [RFC5234].

1.2. Changes from RFC 4627

This section lists all changes between this document and the text in RFC 4627.

- o Applied errata #607 from RFC 4627 to correctly align the artwork for the definition of "object".

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. If a key is duplicated, a parser MAY reject. If it does not reject, then it MUST take only the last of the duplicated key pairs.

```
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

A number is represented in base 10 with no superfluous leading zeroes or punctuation such as commas or spaces. It may have a preceding

minus sign. It may have a "."-prefixed fractional part. It may have an exponent, prefixed by "e" or "E" and optionally "+" or "-".

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 is a sequence of code units wrapped with quotation marks. All 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 Unicode 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 "\\".

```
   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
```

The following four cases MUST all produce the same result:

```
"\u002F"
"\u002F"
"\/"
"/"
```

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". A generator SHOULD NOT emit unpaired surrogates. A parser MAY reject JSON text containing unpaired surrogates.

3. 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.

4. Generators

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

5. Security Considerations

With any data format, it is important to encode correctly. Care must be taken when constructing JSON texts by concatenation. For example:

```
account = 4627;
comment = "\",\"account\":262";   // provided by attacker
json_text = "(\"account\":" + account + ",\"comment\":\"" + comment + "\"}";
```

The result will be

```
{"account":4627,"comment":"","account":262}
```

which some parsers MAY see as being the same as

```
{"comment":"","account":262}
```

This confusion allows an attacker to modify the account property or any other property.

It is much wiser to use JSON generators, which are available in many forms for most programming languages, to do the encoding, avoiding the confusion hazard.

JSON is so similar to some programming languages that the native parsing ability of the language processors can be used to parse JSON texts. This should be avoided because the native parser will accept code which is not JSON.

For example, JavaScript's eval() function is able parse JSON text, but is can also parse programs. If an attacker can inject code into the JSON text (as we saw above), then it can compromise the system. JSON parsers should always be used instead.

The web browser's script tag is an alias for the eval() function. It should not be used to deliver JSON text to web browsers.

6. 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"
  }
```

]

7. Normative References

[ECMA] European Computer Manufacturers Association, "ECMAScript Language Specification Fifth Edition ", December 2009, <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.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[UNICODE] The Unicode Consortium, "The Unicode Standard, Version 6.2 ", 2012, <http://www.unicode.org/versions/Unicode6.2.0/>.

Author's Address

Douglas Crockford
JSON.org

Email: douglas@crockford.com