

JSON Working Group  
Internet-Draft  
Obsoletes: [4627](#) (if approved)  
Intended status: Standards Track  
Expires: June 07, 2014

T. Bray, Ed.  
Google, Inc.  
December 04, 2013

**The JSON Data Interchange Format**  
**draft-ietf-json-rfc4627bis-08**

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.

This document makes no changes to the definition of JSON; it repairs specification errors and offers experience-based interoperability guidance.

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 June 07, 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

<a href="#">1.</a>	Introduction . . . . .	<a href="#">2</a>
<a href="#">1.1.</a>	Conventions Used in This Document . . . . .	<a href="#">3</a>
<a href="#">1.2.</a>	Specifications of JSON . . . . .	<a href="#">3</a>
<a href="#">1.3.</a>	Introduction to This Revision . . . . .	<a href="#">3</a>
<a href="#">2.</a>	JSON Grammar . . . . .	<a href="#">3</a>
<a href="#">3.</a>	Values . . . . .	<a href="#">4</a>
<a href="#">4.</a>	Objects . . . . .	<a href="#">5</a>
<a href="#">5.</a>	Arrays . . . . .	<a href="#">5</a>
<a href="#">6.</a>	Numbers . . . . .	<a href="#">5</a>
<a href="#">7.</a>	Strings . . . . .	<a href="#">7</a>
<a href="#">8.</a>	String and Character Issues . . . . .	<a href="#">8</a>
<a href="#">8.1.</a>	Character Encoding . . . . .	<a href="#">8</a>
<a href="#">8.2.</a>	Unicode Characters . . . . .	<a href="#">8</a>
<a href="#">8.3.</a>	String Comparison . . . . .	<a href="#">9</a>
<a href="#">9.</a>	Parsers . . . . .	<a href="#">9</a>
<a href="#">10.</a>	Generators . . . . .	<a href="#">9</a>
<a href="#">11.</a>	IANA Considerations . . . . .	<a href="#">9</a>
<a href="#">12.</a>	Security Considerations . . . . .	<a href="#">10</a>
<a href="#">13.</a>	Examples . . . . .	<a href="#">10</a>
<a href="#">14.</a>	Contributors . . . . .	<a href="#">12</a>
<a href="#">15.</a>	References . . . . .	<a href="#">12</a>
<a href="#">15.1.</a>	Normative References . . . . .	<a href="#">12</a>
<a href="#">15.2.</a>	Informative References . . . . .	<a href="#">12</a>
<a href="#">Appendix A.</a>	Changes from <a href="#">RFC 4627</a> . . . . .	<a href="#">13</a>
	Author's Address . . . . .	<a href="#">14</a>

## [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-262](#)].

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 [[RFC5234](#)].

### **1.2. Specifications of JSON**

This document is an update of [[RFC4627](#)], which described JSON and registered the Media Type "application/json".

A description of JSON in ECMAScript terms appears in version 5.1 of the ECMAScript specification [[ECMA-262](#)], section 15.12. JSON is also described in [[ECMA-404](#)].

All of the specifications of JSON syntax agree on the syntactic elements of the language.

### **1.3. Introduction to This Revision**

In the years since the publication of [RFC 4627](#), JSON has found very wide use. This experience has revealed certain patterns which, while allowed by its specifications, have caused interoperability problems.

Also, a small number of errata have been reported.

This revision does not change any of the rules of the specification; all texts which were legal JSON remain so, and none which were not JSON become JSON. The revision's goal is to fix the errata and highlight practices which can lead to interoperability problems.

## **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 value. Note that certain previous specifications of JSON constrained a JSON text to be an object or an array. Implementations which generate only objects or arrays where a JSON text is called for will be interoperable in the sense that all implementations will accept these as conforming JSON texts.

JSON-text = ws value ws

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
```

### **3. 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

#### **4. 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
```

An object whose names are all unique is interoperable in the sense that all software implementations which receive that object will agree on the name-value mappings. When the names within an object are not unique, the behavior of software that receives such an object is unpredictable. Many implementations report the last name/value pair only; other implementations report an error or fail to parse the object; other implementations report all of the name/value pairs, including duplicates.

#### **5. 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
```

#### **6. 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 in the grammar below (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
```

This specification allows implementations to set limits on the range and precision of numbers accepted. Since software which implements IEEE 754-2008 binary64 (double precision) numbers [[IEEE754](#)] is generally available and widely used, good interoperability can be achieved by implementations which expect no more precision or range than these provide, in the sense that implementations will approximate JSON numbers within the expected precision. A JSON number such as 1E400 or 3.141592653589793238462643383279 may indicate potential interoperability problems since it suggests that the software which created it expected greater magnitude or precision than is widely available.

Note that when such software is used, numbers which are integers and are in the range  $[-(2^{53})+1, (2^{53})-1]$  are interoperable in the sense that implementations will agree exactly on their numeric values.



## 7. 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 through 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

```

## 8. String and Character Issues

### 8.1. Character Encoding

JSON text SHALL be encoded in Unicode. The default encoding is UTF-8, and JSON texts which are encoded in UTF-8 are interoperable in the sense that they will be read successfully by the maximum number of implementations; there are many implementations which cannot successfully read texts in other encodings (such as UTF-16 and UTF-32).

Implementations have been observed to generate JSON texts prefixed with a Byte Order Mark character. While this is not allowed by the grammar in this specification, in the interests of interoperability it is RECOMMENDED that implementations which parse JSON texts ignore the presence of a byte order mark rather than treating it as an error.

### 8.2. Unicode Characters

When all the strings represented in a JSON text are composed entirely of Unicode characters [[UNICODE](#)] (however escaped), then that JSON text is interoperable in the sense that all software implementations which parse it will agree on the contents of names and of string values in objects and arrays.



However, the ABNF in this specification allows member names and string values to contain bit sequences which cannot encode Unicode characters, for example `"\uDEAD"` (a single unpaired UTF-16 surrogate). Instances of this have been observed, for example when a library truncates a UTF-16 string without checking whether the truncation split a surrogate pair. The behavior of software which receives JSON texts containing such values is unpredictable; for example, implementations might return different values for the length of a string value, or even suffer fatal runtime exceptions.

### **8.3. String Comparison**

Software implementations are typically required to test names of object members for equality. Implementations which transform the textual representation into sequences of Unicode code units, and then perform the comparison numerically, code unit by code unit, are interoperable in the sense that implementations will agree in all cases on equality or inequality of two strings. For example, implementations which compare strings with escaped characters unconverted may incorrectly find that `"a\\b"` and `"a\u005Cb"` are not equal.

## **9. 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 and precision of numbers. An implementation may set limits on the length and character contents of strings.

## **10. Generators**

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

## **11. 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: binary

Interoperability considerations: Described in this document

Published specification: This document

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#, Clojure, ColdFusion, Common Lisp, E, Erlang, Go, Java, JavaScript, Lua, Objective CAML, Perl, PHP, Python, Rebol, Ruby, Scala, 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: IESG  
<iesg@ietf.org>

Intended usage: COMMON

Restrictions on usage: none

Author: Douglas Crockford  
<douglas@crockford.com>

Change controller: IESG  
<iesg@ietf.org>

## **12. Security Considerations**

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

Since JSON's syntax is borrowed from JavaScript, it is possible to use that language's "eval()" function to parse JSON texts. This generally constitutes an unacceptable security risk, since the text could contain executable code along with data declarations. The same consideration applies to the use of eval()-like functions in any other programming language in which JSON texts conform to that language's syntax.

## **13. 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
    },
    "Animated" : false,
    "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"
  }
]
```

Here are three small JSON texts containing only values



"Hello world!"

42

true

## **14. Contributors**

[RFC 4627](#) was written by Douglas Crockford. This document was constructed by making a relatively small number of changes to that document; thus the vast majority of the text here is his.

## **15. References**

### **15.1. Normative References**

- [IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", 2008, <<http://grouper.ieee.org/groups/754/>>.
- [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", 2003-, <<http://www.unicode.org/versions/latest/>>.

Note that this reference is to the latest version of Unicode, rather than to a specific release. It is not expected that future changes in the UNICODE specification will impact the syntax of JSON.

### **15.2. Informative References**

- [ECMA-262] European Computer Manufacturers Association, "ECMAScript Language Specification 5.1 Edition ", June 2011, <<http://www.ecma-international.org/ecma-262/5.1/>>.
- [ECMA-404] Ecma International, "The JSON Data Interchange Format ", October 2013, <<http://www.ecma-international.org/publications/standards/Ecma-404.htm>>.



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

#### **[Appendix A](#). Changes from [RFC 4627](#)**

This section lists changes between this document and the text in [RFC 4627](#).

- o Changed Working Group attribution to JSON Working Group.
- o Changed title and abstract of document.
- o Change the reference to [\[UNICODE\]](#) to be non-version-specific.
- o Added a "Specifications of JSON" section.
- o Added an "Introduction to this Revision" section.
- o Added language about duplicate object member names and interoperability.
- o Applied erratum #607 from [RFC 4627](#) to correctly align the artwork for the definition of "object".
- o Changed "as sequences of digits" to "in the grammar below" in "Numbers" section.
- o Added language about number interoperability as a function of IEEE754, and an IEEE754 reference.
- o Added language about interoperability and Unicode characters, and about string comparisons. To do this, turned the old "Encoding" section into a "String and Character Issues" section, with three subsections: The old "Encoding" material, and two new sections for "Unicode Characters" and "String Comparison".
- o Changed guidance in "Parsers" section to point out that implementations may set limits on the range "and precision" of numbers.
- o Updated and tidied the "IANA Considerations" section.
- o Made a real "Security Considerations" section, and lifted the text out of the existing "IANA Considerations" section.
- o Applied erratum #3607 from [RFC 4627](#) by removing the security consideration that begins "A JSON text can be safely passed" and the JavaScript code that went with that consideration.





- o Added a note to the "Security Considerations" section pointing out the risks of using the "eval()" function in JavaScript or any other language in which JSON texts conform to that language's syntax.
- o Changed "100" to 100 and added a boolean field, both in the first example.
- o Added "Contributors" section crediting Douglas Crockford.
- o Added a reference to [RFC4627](#).
- o Moved the ECMAScript reference from Normative to Informative, updated it to reference ECMAScript 5.1, and added reference to ECMA 404.

#### Author's Address

Tim Bray (editor)  
Google, Inc.

Email: [tbray@textuality.com](mailto:tbray@textuality.com)

