

# The JavaScript Object Notation (JSON) Data Interchange Format

## 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 removes inconsistencies with other specifications of JSON, repairs specification errors, and offers experience-based interoperability guidance.

## Status of this Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in [Section 2 of RFC 7841](#)<sup>1</sup>.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <https://www.rfc-editor.org/info/rfc8259><sup>2</sup>.

## 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 (<https://trustee.ietf.org/license-info><sup>3</sup>) 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.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials,

<sup>1</sup> <https://www.rfc-editor.org/rfc/rfc7841.html#section-2>

<sup>2</sup> <https://www.rfc-editor.org/info/rfc8259>

<sup>3</sup> <https://trustee.ietf.org/license-info>

this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

## Table of Contents

<b>1 Introduction</b>	<b>4</b>
1.1 Conventions Used in This Document	4
1.2 Specifications of JSON	4
1.3 Introduction to This Revision	4
<b>2 JSON Grammar</b>	<b>6</b>
<b>3 Values</b>	<b>7</b>
<b>4 Objects</b>	<b>8</b>
<b>5 Arrays</b>	<b>9</b>
<b>6 Numbers</b>	<b>10</b>
<b>7 Strings</b>	<b>11</b>
<b>8 String and Character Issues</b>	<b>12</b>
8.1 Character Encoding	12
8.2 Unicode Characters	12
8.3 String Comparison	12
<b>9 Parsers</b>	<b>13</b>
<b>10 Generators</b>	<b>14</b>
<b>11 IANA Considerations</b>	<b>15</b>
<b>12 Security Considerations</b>	<b>17</b>
<b>13 Examples</b>	<b>18</b>
<b>14 References</b>	<b>20</b>
14.1 Normative References	20
14.2 Informative References	20
<b>Appendix A Changes from RFC 7159</b>	<b>21</b>
<b>Appendix B Contributors</b>	<b>22</b>
<b>Author's Address</b>	<b>23</b>

## 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 [\[UNICODE\]](#). Note that this citation references the latest version of Unicode rather than a specific release. It is not expected that future changes in the Unicode specification will impact the syntax of JSON.

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", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [\[RFC2119\]](#) [\[RFC8174\]](#) when, and only when, they appear in all capitals, as shown here.

The grammatical rules in this document are to be interpreted as described in [\[RFC5234\]](#).

### 1.2. Specifications of JSON

This document replaces [\[RFC7159\]](#). [\[RFC7159\]](#) obsoleted [\[RFC4627\]](#), which originally described JSON and registered the media type "application/json".

JSON is also described in [\[ECMA-404\]](#).

The reference to ECMA-404 in the previous sentence is normative, not with the usual meaning that implementors need to consult it in order to understand this document, but to emphasize that there are no inconsistencies in the definition of the term "JSON text" in any of its specifications. Note, however, that ECMA-404 allows several practices that this specification recommends avoiding in the interests of maximal interoperability.

The intent is that the grammar is the same between the two documents, although different descriptions are used. If there is a difference found between them, ECMA and the IETF will work together to update both documents.

If an error is found with either document, the other should be examined to see if it has a similar error; if it does, it should be fixed, if possible.

If either document is changed in the future, ECMA and the IETF will work together to ensure that the two documents stay aligned through the change.

### 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 that, while allowed by its specifications, have caused interoperability problems.

Also, a small number of errata have been reported regarding RFC 4627 (see RFC Errata IDs 607 [\[Err607\]](#) and 3607 [\[Err3607\]](#)) and regarding RFC 7159 (see RFC Errata IDs 3915 [\[Err3915\]](#), 4264 [\[Err4264\]](#), 4336 [\[Err4336\]](#), and 4388 [\[Err4388\]](#)).

This document's goal is to apply the errata, remove inconsistencies with other specifications of JSON, and highlight practices that 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 that 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 receiving 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, and some implementations report all of the name/value pairs, including duplicates.

JSON parsing libraries have been observed to differ as to whether or not they make the ordering of object members visible to calling software. Implementations whose behavior does not depend on member ordering will be interoperable in the sense that they will not be affected by these differences.



## 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
```

There is no requirement that the values in an array be of the same type.

## 6. Numbers

The representation of numbers is similar to that used in most programming languages. A number is represented in base 10 using decimal digits. It 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. 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 uppercase 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 that implements IEEE 754 binary64 (double precision) numbers [\[IEEE754\]](#) is generally available and widely used, good interoperability can be achieved by implementations that 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 that created it expects receiving software to have greater capabilities for numeric magnitude and precision than is widely available.

Note that when such software is used, numbers that 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 uppercase 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 12-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 exchanged between systems that are not part of a closed ecosystem **MUST** be encoded using UTF-8 [\[RFC3629\]](#).

Previous specifications of JSON have not required the use of UTF-8 when transmitting JSON text. However, the vast majority of JSON-based software implementations have chosen to use the UTF-8 encoding, to the extent that it is the only encoding that achieves interoperability.

Implementations **MUST NOT** add a byte order mark (U+FEFF) to the beginning of a networked-transmitted JSON text. In the interests of interoperability, implementations that parse JSON texts **MAY** 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 that 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 that 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 that 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 that 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 that 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 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
Security considerations:	See RFC 8259, Section 12
Interoperability considerations:	Described in RFC 8259
Published specification:	RFC 8259
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,

Additional information:

Person & email address to contact for further information:

Intended usage:

Restrictions on usage:

Author:

Change controller:

Note:

Objective  
CAML,  
Perl,  
PHP,  
Python,  
Rebol,  
Ruby,  
Scala,  
and  
Scheme.

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

IESG  
<iesg@ietf.org>

COMMON

none

Douglas  
Crockford  
<douglas@crockford.com>

IESG  
<iesg@ietf.org>

No  
"charset"  
parameter  
is  
defined  
for  
this  
registration.  
Adding  
one  
really  
has  
no  
effect  
on  
compliant  
recipients.



## 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 most JSON texts (but not all; certain characters such as U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR are legal in JSON but not JavaScript). 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. References

### 14.1. Normative References

- [ECMA-404] Ecma International, "[The JSON Data Interchange Format](http://www.ecma-international.org/publications/standards/Ecma-404.htm)", Standard ECMA-404, <<http://www.ecma-international.org/publications/standards/Ecma-404.htm>>.
- [IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754.
- [RFC2119] Bradner, S., "[Key words for use in RFCs to Indicate Requirement Levels](https://www.rfc-editor.org/info/rfc2119)", BCP 14, RFC 2119, DOI [10.17487/RFC2119](https://doi.org/10.17487/RFC2119), March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3629] Yergeau, F., "[UTF-8, a transformation format of ISO 10646](https://www.rfc-editor.org/info/rfc3629)", STD 63, RFC 3629, DOI [10.17487/RFC3629](https://doi.org/10.17487/RFC3629), November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "[Augmented BNF for Syntax Specifications: ABNF](https://www.rfc-editor.org/info/rfc5234)", STD 68, RFC 5234, DOI [10.17487/RFC5234](https://doi.org/10.17487/RFC5234), January 2008, <<https://www.rfc-editor.org/info/rfc5234>>.
- [RFC8174] Leiba, B., "[Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words](https://www.rfc-editor.org/info/rfc8174)", BCP 14, RFC 8174, DOI [10.17487/RFC8174](https://doi.org/10.17487/RFC8174), May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [UNICODE] The Unicode Consortium, "[The Unicode Standard](http://www.unicode.org/versions/latest/)", <<http://www.unicode.org/versions/latest/>>.

### 14.2. Informative References

- [ECMA-262] Ecma International, "[ECMAScript Language Specification](http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf)", Standard ECMA-262, Third Edition, December 1999, <<http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf>>.
- [Err3607] RFC Errata, "[Erratum ID 3607](https://www.rfc-editor.org/errata/eid3607)", RFC 4627, <<https://www.rfc-editor.org/errata/eid3607>>.
- [Err3915] RFC Errata, "[Erratum ID 3915](https://www.rfc-editor.org/errata/eid3915)", RFC 7159, <<https://www.rfc-editor.org/errata/eid3915>>.
- [Err4264] RFC Errata, "[Erratum ID 4264](https://www.rfc-editor.org/errata/eid4264)", RFC 7159, <<https://www.rfc-editor.org/errata/eid4264>>.
- [Err4336] RFC Errata, "[Erratum ID 4336](https://www.rfc-editor.org/errata/eid4336)", RFC 7159, <<https://www.rfc-editor.org/errata/eid4336>>.
- [Err4388] RFC Errata, "[Erratum ID 4388](https://www.rfc-editor.org/errata/eid4388)", RFC 7159, <<https://www.rfc-editor.org/errata/eid4388>>.
- [Err607] RFC Errata, "[Erratum ID 607](https://www.rfc-editor.org/errata/eid607)", RFC 4627, <<https://www.rfc-editor.org/errata/eid607>>.
- [RFC4627] Crockford, D., "[The application/json Media Type for JavaScript Object Notation \(JSON\)](https://www.rfc-editor.org/info/rfc4627)", RFC 4627, DOI [10.17487/RFC4627](https://doi.org/10.17487/RFC4627), July 2006, <<https://www.rfc-editor.org/info/rfc4627>>.
- [RFC7159] Bray, T., Ed., "[The JavaScript Object Notation \(JSON\) Data Interchange Format](https://www.rfc-editor.org/info/rfc7159)", RFC 7159, DOI [10.17487/RFC7159](https://doi.org/10.17487/RFC7159), March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.

## Appendix A. Changes from RFC 7159

This section lists changes between this document and the text in RFC 7159.

- [Section 1.2](#) has been updated to reflect the removal of a JSON specification from ECMA-262, to make ECMA-404 a normative reference, and to explain the particular meaning of "normative".
- [Section 1.3](#) has been updated to reflect errata filed against RFC 7159, not RFC 4627.
- [Section 8.1](#) was changed to require the use of UTF-8 when transmitted over a network.
- [Section 12](#) has been updated to increase the precision of the description of the security risk that follows from using the ECMAScript "eval()" function.
- [Section 14.1](#) has been updated to include ECMA-404 as a normative reference.
- [Section 14.2](#) has been updated to remove ECMA-404, update the version of ECMA-262, and refresh the errata list.

## Appendix B. 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.

## **Author's Address**

**Tim Bray** (editor)

Textuality

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