JSON

From Wikipedia, the free encyclopedia

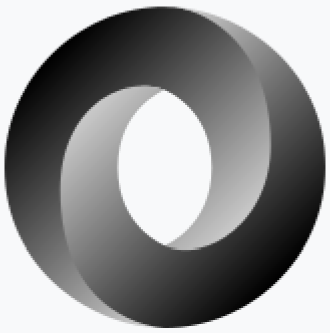
In computing, **JSON** (canonically pronounced /ˈdʒeɪsən/ *JAY-sən*;[1] sometimes **JavaScript Object Notation**) is an open-standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is the most common data format used for asynchronous browser/server communication, largely replacing XML, and is used by AJAX.

JSON is a language-independent data format. It was derived from JavaScript, but as of 2017 many programming languages include code to generate and parse JSON-format data. The official Internet media type for JSON is `application/json`. JSON filenames use the extension `.json`.

Douglas Crockford originally specified the JSON format in the early 2000s; two competing standards, RFC 7159 and ECMA-404 (https://www.ecma-international.org/publications/standards/Ecma-404.htm), defined it in 2013. The ECMA standard describes only the allowed syntax, whereas the RFC covers some security and interoperability considerations.[2]

RFC 7493 defines a restricted profile of JSON, known as I-JSON (short for "Internet JSON"), which seeks to overcome some of the interoperability problems with JSON. Every I-JSON document is a valid JSON document but not every valid JSON document is a valid I-JSON document.[3]

JSON



Filename extension	`.json`
Internet media type	`application/json`
Type code	TEXT
Uniform Type Identifier (UTI)	public.json
Type of format	Data interchange
Extended from	JavaScript
Standard	RFC 7159, ECMA-404 (http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf)
Website	json.org (http://json.org/)

Contents

History

JSON grew out of a need for stateful, real-time server-to-browser communication protocol without using browser plugins such as Flash or Java applets, the dominant methods used in the early 2000s.

Douglas Crockford first specified JSON. Image from 2007.

Douglas Crockford first specified[4] and popularized the JSON format. The acronym originated at State Software (originally named Veil Networks, Inc.), a company co-founded by Crockford, F. Randall "Randy" Farmer (left to work at 3DO), Greg Macdonald, Chip Morningstar, Robert F. Napiltonia and Dominik Zynis in March 2001. State Software was funded by Tesla Ventures with $1.8 Million in October 2011, and attempted to trademark the word "State". The co-founders agreed to build a system that used standard browser capabilities and provided an abstraction layer for Web developers to create stateful Web applications that had a persistent duplex connection to a Web server by holding two HTTP connections open and recycling them before standard browser time-outs if no further data were exchanged. The co-founders had a round-table discussion and voted whether to call the data format JSML or JSON, as well as under what license type to make it available. Crockford, being inspired by the words of then President Bush, should also be credited with coming up with the "evil-doers" JSON license ("The Software shall be used for Good, not Evil.") in order to open-source the JSON libraries, but force (troll) corporate lawyers, or those who are overly pedantic, to seek to pay for a license from State. Morningstar developed the idea for the State Application Framework at State Software.[5][6] On the other hand, this clause lead to license compatibility problems of the JSON license with other open-source licenses.[7]

A precursor to the JSON libraries was used in a children's digital asset trading game project named Cartoon Orbit at Communities.com (the State co-founders had all worked at this company previously) for Cartoon Network, which used a browser side plug-in with a proprietary messaging format to manipulate DHTML elements (this system is also owned by 3DO). Upon discovery of early Ajax capabilities, digiGroups, Noosh, and others used frames to pass information into the user browsers' visual field without refreshing a Web application's visual context, realizing real-time rich Web applications using only the standard HTTP, HTML and JavaScript capabilities of Netscape 4.0.5+ and IE 5+. Crockford then found that JavaScript could be used as an object-based messaging format for such a system. The system was sold to Sun Microsystems, Amazon.com and EDS. The JSON.org (http://json.org/) Web site was launched in 2002. In December 2005, Yahoo! began offering some of its Web services in JSON.[8] Google started offering JSON feeds for its GData web protocol in December 2006.[9]

JSON was originally intended to be a subset of the JavaScript scripting language (specifically, Standard ECMA-262 3rd Edition—December 1999[10]) and is commonly used with Javascript, but it is a language-independent data format. Code for parsing and generating JSON data is readily available in many programming languages. JSON's Web site lists JSON libraries by language.

Though JSON was originally advertised and believed to be a strict subset of JavaScript and ECMAScript,[11] it inadvertently allows some unescaped characters in strings that are illegal in JavaScript and ECMAScript strings. See Data portability issues below.

JSON itself became an ECMA international standard in 2013 as the *ECMA-404 standard*.[12] In the same year RFC 7158 used ECMA-404 as reference. In 2014 RFC 7159 became the main reference for JSON's internet uses (ex. MIME application/json), and obsoletes RFC 4627 and RFC 7158 (but preserving ECMA-262 and ECMA-404 as main references).

Data types, syntax and example

JSON's basic data types are:

- Number: a signed decimal number that may contain a fractional part and may use exponential E notation, but cannot include non-numbers like NaN. The format makes no distinction between integer and floating-point. JavaScript uses a double-precision floating-point format for all its numeric values, but other languages implementing JSON may encode numbers differently.
- String: a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.
- Boolean: either of the values `true` or `false`
- Array: an ordered list of zero or more values, each of which may be of any type. Arrays use square bracket notation with elements being comma-separated.
- Object: an unordered collection of name/value pairs where the names (also called keys) are strings. Since objects are intended to represent associative arrays,[12] it is recommended, though not required,[13] that each key is unique within an object. Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value.
- `null`: An empty value, using the word `null`

Limited whitespace is allowed and ignored around or between syntactic elements (values and punctuation, but not within a string value). Only four specific characters are considered whitespace for this purpose: space, horizontal tab, line feed, and carriage return. In particular, the byte order mark must not be generated by a conforming implementation (though it may be accepted when parsing JSON). JSON does not provide any syntax for comments.

Early versions of JSON (such as specified by RFC 4627) required that a valid JSON "document" must consist of only an object or an array type, which could contain other types within them. This restriction was removed starting with RFC 7158, so that a JSON document may consist entirely of any possible JSON typed value. (Note that RFC 7159, dated March 2014, now obsoletes both 4627 and 7158.)

Example

The following example shows a possible JSON representation describing a person.

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

Data portability issues

Although Douglas Crockford originally claimed that JSON is a strict subset of JavaScript, his specification actually allows valid JSON documents that are invalid JavaScript. Specifically, JSON allows the Unicode line terminators `U+2028` LINE SEPARATOR and `U+2029` PARAGRAPH SEPARATOR to appear unescaped in quoted strings, while JavaScript does not.[14] This is a consequence of JSON disallowing only "control characters". For maximum portability, these characters should be backslash-escaped. This subtlety is important when generating JSONP.

JSON documents can be encoded in UTF-8, UTF-16 or UTF-32, the default encoding being UTF-8.[13] These encodings support the full Unicode character set, including those characters outside the Basic Multilingual Plane (U+10000 to U+10FFFF). However, if escaped, those characters must be written using UTF-16 surrogate pairs, a detail missed by some JSON parsers. For example, to include the Emoji character `U+1F602` 😂 FACE WITH TEARS OF JOY in JSON:

```
{ "face": "😂" }
// or
{ "face": "\uD83D\uDE02" }
```

Numbers in JSON are agnostic with regard to their representation within programming languages. No differentiation is made between an integer and floating-point value: some implementations may treat `42`, `42.0`, and `4.2E+1` as the same number while others may not. Furthermore, no requirements are made regarding implementation issues such as overflow, underflow, loss of precision, or rounding. Additionally, JSON says nothing about the treatment of signed zeros: whether `0.0` is distinct from `-0.0`. Most implementations that use the IEEE 754 floating-point standard, including JavaScript, preserve signed zeros; but not all JSON implementations may do so.

Using JSON in JavaScript

Since JSON was derived from JavaScript and its syntax is (mostly) a subset of the language, it is possible to use the JavaScript `eval()` function to parse JSON data. Due to the issue with parsing of Unicode line terminators discussed in the preceding section, the eval function needs to perform a string replacement.[14]

```
var p = eval('(' + json_string.replace(/\u2028/g,'\\u2028').replace(/\u2029/g, '\\u2029') + ')');
```

This is unsafe if the string is untrusted. Instead, a JSON parser library or JavaScript's native JSON support should be used for reading and writing JSON.

```
var p = JSON.parse(json_string);
```

A correctly implemented JSON parser only accepts valid JSON, preventing potentially malicious code from being inadvertently executed.

Since 2010, web browsers such as Firefox and Internet Explorer have included support for parsing JSON. As native browser support is more efficient and secure than `eval()`, native JSON support is included in Edition 5 of the ECMAScript standard.[15]

Unsupported native data types

JavaScript syntax defines several native data types that are not included in the JSON standard:[13] Map, Set, Date, Error, Regular Expression, Function, Promise, and `undefined`.[note 1] These JavaScript data types must be represented by some other data format, with the programs on both ends agreeing on how to convert between the types. As of 2011, there are some de facto standards, *e.g.*, converting from Date to String, but none universally recognized.[16][17] Other languages may have a different set of native types that must be serialized carefully to deal with this type of conversion.

Schema and metadata

JSON Schema

JSON Schema[18] specifies a JSON-based format to define the structure of JSON data for validation, documentation, and interaction control. A JSON Schema provides a contract for the JSON data required by a given application, and how that data can be modified.

JSON Schema is based on the concepts from XML Schema (XSD), but is JSON-based. The JSON data schema can be used to validate JSON data. As in XSD, the same serialization/deserialization tools can be used both for the schema and data. The schema is self-describing.

JSON Schema is an Internet Draft currently in its 5th version, which was released on October 13, 2016[19] (version 4 had expired on August 4, 2013,[20] leading to a lapse of more than 3 years). There are several validators available for different programming languages,[21] each with varying levels of conformance. There are no standard file extension, but some suggested `.schema.json`.[22]

Example JSON Schema (draft 4):

```
{
  "$schema": "http://json-schema.org/schema#",
  "title": "Product",
  "type": "object",
  "required": ["id", "name", "price"],
  "properties": {
    "id": {
      "type": "number",
      "description": "Product identifier"
    },
    "name": {
      "type": "string",
      "description": "Name of the product"
    },
    "price": {
      "type": "number",
      "minimum": 0
    },
    "tags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "stock": {
      "type": "object",
      "properties": {
        "warehouse": {
          "type": "number"
        },
        "retail": {
          "type": "number"
        }
      }
    }
  }
}
```

The JSON Schema above can be used to test the validity of the JSON code below:

```
{
  "id": 1,
  "name": "Foo",
  "price": 123,
  "tags": [
    "Bar",
    "Eek"
  ],
  "stock": {
    "warehouse": 300,
    "retail": 20
  }
}
```

MIME type

The official MIME type for JSON text is "`application/json`".[23] Although most modern implementations have adopted the official MIME type, many applications continue to provide legacy support for other MIME types. Many service providers, browsers, servers, web applications, libraries, frameworks, and APIs use, expect, or recognize the (unofficial) MIME type "`text/json`" or the content-type "`text/javascript`". Notable examples include the Google Search API,[24] Yahoo!,[24][25] Flickr,[24] Facebook API,[26] Lift framework,[27] Dojo Toolkit 0.4,[28] etc.

Applications

JSON-RPC

JSON-RPC is a remote procedure call (RPC) protocol built on JSON, as a replacement for XML-RPC or SOAP. It is a simple protocol that defines only a handful of data types and commands. JSON-RPC lets a system send notifications (information to the server that does not require a response) and multiple calls to the server that can be answered out of order. Example of a JSON-RPC 2.0 request and response using positional parameters.

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

AJAJ

Asynchronous JavaScript and JSON (or AJAJ) refers to the same dynamic web page methodology as Ajax, but instead of XML, JSON is the data format. AJAJ is a web development technique that provides for the ability of a webpage to request new data after it has loaded into the web browser. Typically it renders new data from the server in response to user actions on that webpage. For example, what the user types into a search box, client-side code then sends to the server, which immediately responds with a drop-down list of matching database items.

The following JavaScript code is an example of a client using XMLHttpRequest to request data in JSON format from a server. (The server-side programming is omitted; it must be set up to service requests to the `url` containing a JSON-formatted string.)

```
var my_JSON_object;
var http_request = new XMLHttpRequest();
http_request.open("GET", url, true);
http_request.responseType = "json";
http_request.onreadystatechange = function () {
  var done = 4, ok = 200;
  if (http_request.readyState === done && http_request.status === ok) {
    my_JSON_object = http_request.response;
  }
};
http_request.send(null);
```

Security issues

JSON is intended as a data serialization format. However, its design as a non-strict subset of the JavaScript scripting language poses several security concerns. These concerns center on the use of a JavaScript interpreter to execute JSON text dynamically as embedded JavaScript. This exposes a program to errant or malicious scripts. This is a serious issue when dealing with data retrieved from the Internet. This easy and popular but risky technique exploits JSON's compatibility with the JavaScript `eval()` function, which is described below.

JavaScript `eval()`

Some developers mistakenly believe that JSON-formatted text is also syntactically legal JavaScript code though certain valid JSON strings are actually not valid JavaScript code.[29] Because of that, they believe an easy way for a JavaScript program to parse JSON-formatted data would be to use the built-in JavaScript `eval()` function, which was designed to evaluate JavaScript expressions. Rather than using a JSON-specific parser, the JavaScript interpreter itself is thus used to execute the JSON data, producing native JavaScript objects. This technique is risky, however, if there is any chance that the JSON data might contain arbitrary JavaScript code, which would then be executed also.

Unless precautions are taken to validate the data first, the eval technique is subject to security vulnerabilities when the data and the entire JavaScript environment are not within the control of a single trusted source. For example, if the data is itself not trusted, it is subject to malicious JavaScript code injection attacks. Such breaches of trust also can create vulnerabilities for data theft, authentication forgery, and other potential misuses of data and resources.

The RFC that defines JSON (RFC 4627) suggests using the following code to validate JSON using a regular expression before evaluating it (the variable 'text' is the input JSON):[30]

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

However, this validation is now known to be insufficient.[31]

A new function, `JSON.parse()`, was thus developed as a safer alternative to `eval()`. It is specifically intended to process JSON data and not JavaScript. It was originally planned for inclusion in the Fourth Edition of the ECMAScript standard,[32] but this did not occur. It was first added to the Fifth Edition,[33] and is now supported by the major browsers given below. For older ones, a compatible JavaScript library is available at JSON.org.

An additional issue when parsing JSON using the `eval()` function is that there are some Unicode characters that are valid in JSON strings but invalid in JavaScript, so additional escaping may be needed in some cases.[29]

Native encoding and decoding in browsers

Web browsers now have or plan to have native JSON encoding and decoding. This eliminates the `eval()` security problem above, and can increase performance compared to the JavaScript libraries commonly used before. As of June 2009, the following browsers have native JSON support, via `JSON.parse()` and `JSON.stringify()`:

- Mozilla Firefox 3.5+[34]
- Microsoft Internet Explorer 8+[35]
- Opera 10.5+[36]
- WebKit-based browsers (Apple Safari)[37]
- Blink-based browsers (e.g. Google Chrome, Opera)

At least five popular JavaScript libraries have committed to use native JSON, if available:

- YUI Library[38]
- Prototype[39]
- jQuery[40]
- Dojo Toolkit[41]
- MooTools[42]

Implementation-specific issues

Various JSON parser implementations suffered in the past from denial-of-service attack and mass assignment vulnerability.[43][44]

Object references

The JSON standard does not support object references, but an IETF draft standard for JSON-based object references exists.[45] The Dojo Toolkit supports object references using standard JSON; specifically, the `dojox.json.ref` module provides support for several forms of referencing including circular, multiple, inter-message, and lazy referencing.[46][47][48] Alternatively, non-standard solutions exist such as the use of Mozilla JavaScript Sharp Variables. However this functionality became obsolete with JavaScript 1.8.5 and was removed in Firefox version 12.[49]

Comparison with other formats

JSON is promoted as a low-overhead alternative to XML as both of these formats have widespread support for creation, reading, and decoding in the real-world situations where they are commonly used.[50] Apart from XML, examples could include OGDL, YAML and CSV. Also, Google Protocol Buffers can fill this role, although it is not a data interchange language.

YAML

YAML version 1.2 is a superset of JSON; prior versions were "not strictly compatible". For example, escaping a slash (/) with a backslash (\) is valid in JSON, but was not valid in YAML. (This is common practice when injecting JSON into HTML to protect against cross-site scripting attacks.) Nonetheless, many YAML parsers can natively parse the output from many JSON encoders.[51]

XML

XML has been used to describe structured data and to serialize objects. Various XML-based protocols exist to represent the same kind of data structures as JSON for the same kind of data interchange purposes. Data can be encoded in XML several ways. The most expansive form using tag pairs results in a much larger representation than JSON, but if data is stored in attributes and 'short tag' form where the closing tag is replaced with '/>', the representation is often about the same size as JSON or just a little larger. If the data is compressed using an algorithm like gzip, there is little difference because compression is good at saving space when a pattern is repeated.

XML also has the concept of schema. This permits strong typing, user-defined types, predefined tags, and formal structure, allowing for formal validation of an XML stream in a portable way. There is, however, an IETF draft proposal for a schema system for JSON [1] (http://json-schema.org/documentation.html).

XML supports comments, but JSON does not.[52]

Samples

JSON sample

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

Both of the following examples carry the same kind of information as the JSON example above in different ways. There is a powerful jq language and tool to deal with JSON.[53][54][55]

YAML sample

The JSON code above is also entirely valid YAML. YAML also offers an alternative syntax intended to be more human-accessible by replacing nested delimiters like `{}`, `[]`, and `"` marks with off-side indentation.[51]

```
---
firstName: John
lastName: Smith
age: 25
address:
  streetAddress: 21 2nd Street
  city: New York
  state: NY
  postalCode: '10021'
phoneNumber:
- type: home
  number: 212 555-1234
- type: fax
```

```
  number: 646 555-4567
gender:
  type: male
```

XML samples

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>
      <type>home</type>
      <number>212 555-1234</number>
    </phoneNumber>
    <phoneNumber>
      <type>fax</type>
      <number>646 555-4567</number>
    </phoneNumber>
  </phoneNumbers>
  <gender>
    <type>male</type>
  </gender>
</person>
```

The properties can also be serialized using attributes instead of tags:

```
<person firstName="John" lastName="Smith" age="25">
  <address streetAddress="21 2nd Street" city="New York" state="NY" postalCode="10021" />
  <phoneNumbers>
    <phoneNumber type="home" number="212 555-1234"/>
    <phoneNumber type="fax" number="646 555-4567"/>
  </phoneNumbers>
  <gender type="male"/>
</person>
```

The XML encoding *may* therefore be comparable in length to the equivalent JSON encoding. A wide range of XML processing technologies exist, from the Document Object Model to XPath and XSLT. XML can also be styled for immediate display using CSS. XHTML is a form of XML so that elements can be passed in this form ready for direct insertion into webpages using client-side scripting.

See also

- JSON Streaming
- Other formats
  - HOCON—Human-Optimized Config Object Notation, a superset of JSON
  - YAML—Another datastorage format that is a superset of JSON[56]
  - S-expression—the comparable LISP format for trees as text.
  - JSONP—JSON with Padding, a pattern of usage commonly employed when retrieving JSON across domains
  - GeoJSON—an open format for encoding a variety of geographic data structures
  - JSON-LD—JavaScript Object Notation for Linked Data, currently a W3C Recommendation
  - JSON-RPC
  - SOAPjr—a hybrid of SOAP and JR (JSON-RPC)
  - JsonML
- Binary encodings for JSON
  - BSON

  - MessagePack
  - Smile
  - UBJSON
  - EXI4JSON (EXI for JSON)—representation by means of the Efficient XML Interchange (EXI) standard
- Implementations:
  - Jayrock—an open source implementation of JSON for the .NET Framework.
  - Ember data server implementations for PHP, Node.js, Ruby, Python, Go, .NET and Java.
- Other
  - Comparison of data serialization formats
  - Jq (programming language)

Notes

1. The `undefined` type was left out of the JSON standard, and one finds suggestions that `null` be used instead. In fact, the current standard says that for a sparse array such as:

```
var v = [0];
v[3] = 3;
```

which behaves in JavaScript as if it were:

```
var vx = [0, undefined, undefined, 3];
```

with the `undefined` entries being only implicit rather than explicit, should translate to JSON as if it were:

```
var vx = [0, null, null, 3];
```

with explicit `null` fillers for the undefined entries.

Furthermore, in JavaScript `{a: undefined}` often behaves the same as `{}`. Both translate as `"{}"` in JSON. However `undefined` as an explicit property value does have use in JavaScript inheritance situations such as:

```
var x = {a: 1};
var xi = Object.create(x);
xi.a = undefined;
```

where the inheritance of `x`'s property `a` is overridden in `xi` and makes it pretty much behave as if nothing was inherited. `JSON.stringify` itself ignores inherited values - it only translates the enumerable own properties as given by `Object.keys(y)`. The default stringification, while not encoding inheritance, can (except for `undefined` values) encode enough of an object to reconstruct it in an environment that knows what inheritance it should have. To encode JavaScript objects that contain explicit `undefined` values a convention for representing `undefined` must be established, such as mapping it to the string `"UNDEFINED"`. One can then pass `JSON.stringify` the optional `replacer` argument to translate with this convention:

```
var y = {a: undefined};
var ys = JSON.stringify(y,
 function (k, v){return (v === undefined) ? "UNDEFINED" : v});
```

Converting this JSON back into JavaScript is not as straightforward. While `JSON.parse` can take an optional `reviver` argument that is, essentially, the inverse of a `replacer`, it can't be used in this situation. If that function returns `undefined`, the `JSON.parse` logic interprets this to mean to not define a property rather than define one with a `undefined` value. Instead one has to explicitly post process the result from `JSON.parse` replacing each `"UNDEFINED"` with `undefined`.

References

1. "Doug Crockford "Google Tech Talks: JavaScript: The Good Parts" ". 7 February 2009.
2. Bray, Tim. "JSON Redux AKA RFC7159". *Ongoing*. Retrieved 16 March 2014.
3. Bray, Tim (ed.), *The I-JSON Message Format*, Internet Engineering Task Force (IETF), RFC 7493
4. "Douglas Crockford — The JSON Saga". YouTube. 28 August 2011. Retrieved 23 September 2016.
5. "Chip Morningstar Biography". n.d.
6. "State Software Breaks Through Web App Development Barrier With State Application Framework: Software Lets Developers Create Truly Interactive Applications; Reduces Costs, Development Time and Improves User Experience". *PR Newswire*. February 12, 2002.
7. Apache and the JSON license (https://lwn.net/Articles/707510/) on LWN.net by Jake Edge (November 30, 2016)
8. Yahoo!. "Using JSON with Yahoo! Web services". Archived from the original on October 11, 2007. Retrieved July 3, 2009.
9. Google. "Using JSON with Google Data APIs". Retrieved July 3, 2009.
10. Crockford, Douglas (May 28, 2009). "Introducing JSON". json.org. Retrieved July 3, 2009.
11. Douglas Crockford (2016-07-10). "JSON in JavaScript". Retrieved 2016-08-13.
12. "The JSON Data Interchange Format" (PDF). ECMA International. October 2013. Retrieved 23 September 2016.
13. "JSON Web Token (JWT)". IETF. May 2015. Retrieved 23 September 2016.
14. Holm, Magnus (15 May 2011). "JSON: The JavaScript subset that isn't". The timeless repository. Retrieved 23 September 2016.
15. "Standard ECMA-262". *ecma-international.org*. Retrieved 13 September 2015.
16. "jquery - Format a Microsoft JSON date? - Stack Overflow". *stackoverflow.com*. Retrieved 13 September 2015.
17. "Tales from the Evil Empire - Dates and JSON". *asp.net*. Retrieved 13 September 2015.
18. "JSON Schema and Hyper-Schema". *json-schema.org*. Retrieved 13 September 2015.
19. "draft-wright-json-schema-00 - JSON Schema: A Media Type for Describing JSON Documents". *json-schema.org/*. Retrieved 17 February 2017.
20. "draft-zyp-json-schema-04 - JSON Schema: core definitions and terminology". *ietf.org*. Retrieved 17 March 2016.
21. "JSON Schema Software". *json-schema.org*. Retrieved 13 September 2015.
22. http://stackoverflow.com/a/10507586/287948
23. "Media Types". *iana.org*. Retrieved 13 September 2015.
24. "Handle application/json & text/json by benschwarz · Pull Request #2 · mislav/faraday-stack". *GitHub*. Retrieved 13 September 2015.
25. "Yahoo!, JavaScript, and JSON". *ProgrammableWeb*. Retrieved 13 September 2015.
26. "Make JSON requests allow text/javascript content by jakeboxer · Pull Request #148 · AFNetworking/AFNetworking". *GitHub*. Retrieved 13 September 2015.
27. "lift/Req.scala at master · lift/lift · GitHub". *GitHub*. Retrieved 13 September 2015.
28. "BrowserIO.js in legacy/branches/0.4/src/io – Dojo Toolkit". *dojotoolkit.org*. Retrieved 13 September 2015.
29. "JSON: The JavaScript subset that isn't". Magnus Holm. Retrieved 16 May 2011.
30. Douglas Crockford (July 2006). "IANA Considerations" (https://tools.ietf.org/html/rfc4627#section-6). *The application/json Media Type for JavaScript Object Notation (JSON)* (https://tools.ietf.org/html/rfc4627). IETF. sec. 6. RFC 4627. https://tools.ietf.org/html/rfc4627#section-6. Retrieved October 21, 2009.
31. Stefano Di Paola. "Minded Security Blog: Ye Olde Crockford JSON regexp is Bypassable". *mindedsecurity.com*. Retrieved 13 September 2015.
32. Crockford, Douglas (December 6, 2006). "JSON: The Fat-Free Alternative to XML". Retrieved July 3, 2009.
33. "ECMAScript Fifth Edition" (PDF). Retrieved March 18, 2011.
34. "Using Native JSON". June 30, 2009. Retrieved July 3, 2009.
35. Barsan, Corneliu (September 10, 2008). "Native JSON in IE8". Retrieved July 3, 2009.
36. "Web specifications supported in Opera Presto 2.5". March 10, 2010. Retrieved March 29, 2010.
37. Hunt, Oliver (June 22, 2009). "Implement ES 3.1 JSON object". Retrieved July 3, 2009.
38. "YUI 2: JSON utility". September 1, 2009. Retrieved October 22, 2009.
39. "Learn JSON". April 7, 2010. Retrieved April 7, 2010.
40. "Ticket #4429". May 22, 2009. Retrieved July 3, 2009.
41. "Ticket #8111". June 15, 2009. Retrieved July 3, 2009.
42. "Ticket 419". October 11, 2008. Retrieved July 3, 2009.
43. "Denial of Service and Unsafe Object Creation Vulnerability in JSON (CVE-2013-0269)". Retrieved January 5, 2016.
44. "Microsoft .NET Framework JSON Content Processing Denial of Service Vulnerability". Retrieved January 5, 2016.
45. Zyp, Kris (September 16, 2012). Bryan, Paul C., ed. "JSON Reference: draft-pbryan-zyp-json-ref-03". *Internet Engineering Task Force*.
46. Zyp, Kris. "dojox.json.ref". *Dojo*.
47. Zyp, Kris (June 17, 2008). "JSON referencing in Dojo". *SitePen*. Retrieved July 3, 2009.

48. von Gaza, Tys (Dec 7, 2010). "JSON referencing in jQuery". *NUBUNTU*. Archived from the original on May 7, 2015. Retrieved Dec 7, 2010.
49. "Sharp variables in JavaScript". *Mozilla Developer Network*. April 4, 2015. Retrieved 21 April 2012.
50. "JSON: The Fat-Free Alternative to XML". json.org. Retrieved 14 March 2011.
51. "YAML Ain't Markup Language (YAML™) Version 1.2". *yaml.org*. Retrieved 13 September 2015.
52. Saternos, Casimir (2014). *Client-server web apps with Javascript and Java*. p. 45. ISBN 9781449369316.
53. Janssens has a couple of examples showing the use of *jq*: Janssens, Jeroen (2014). *Data Science at the Command Line*. O'Reilly Media. ISBN 978-1-4919-4785-2.
54. stackoverflow has a lot of Q&A regarding *jq*: "jq Q&A on stackoverflow". *stackoverflow.com*.
55. *jq* is open source; ready usable executables, documentation and a tutorial are available: Stephen Dolan. "jq is a lightweight and flexible command-line JSON processor". *github.io*. Retrieved 2016-02-27.
56. Oren Ben-Kiki; Clark Evans; Ingy döt Net. "YAML Ain't Markup Language (YAML™) Version 1.2". Retrieved 29 August 2015.

External links

- Format home page (http://www.json.org/)
- The JavaScript Object Notation (JSON) Data Interchange Format (RFC 7159) (https://tools.ietf.org/html/rfc7159)
- ECMA-404 (http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf)—The JSON Data Interchange Format
- CBOR (http://tools.ietf.org/html/rfc7049)—A binary encoding for JSON

Retrieved from "https://en.wikipedia.org/w/index.php?title=JSON&oldid=770890903"

Categories: 2001 introductions | Ajax (programming) | Data serialization formats | JavaScript | JSON | Markup languages | Open formats