JSON: The Fat-Free Alternative to XML

Presented at XML 2006 in Boston, December 6 by Douglas Crockford

In 2005, the idea of *Dynamic HTML* gained new energy and acceptance when it was renamed as *Ajax*. One of the key ideas in Ajax is the use of data interchange as a more efficient alternative to page replacement. Applications are delivered as HTML pages. How should data be delivered?

Historically, we have seen an evolution in data formats, from completely ad hoc methods to techniques that are inspired by database models and document models. The format that will be described here is based on a programming language model, avoiding the obvious brittleness of the ad hoc methods and the impedance mismatch inefficiencies of the other models.

JSON (or JavaScript Object Notation) is a programming language model data interchange format. It is minimal, textual, and a subset of JavaScript. Specifically, it is a subset of ECMA-262 (The ECMAScript programming Language Standard, Third Edition, December 1999). It is lightweight and very easy to parse.

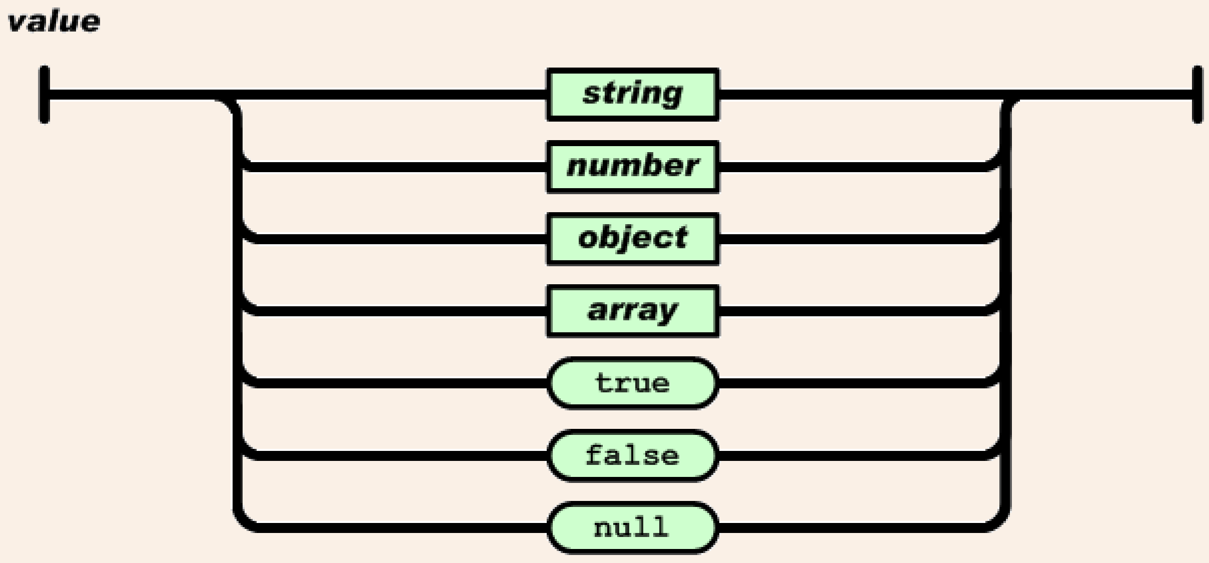
JSON is not a document format. It is not a markup language. It is not even a general serialization format in that it does not have a direct representation for cyclical structures, although it can support a meta representation that does.

A number of people independently discovered that JavaScript's object literals were an ideal format for transmitting object-oriented data across the network. I made my own discovery in April of 2001 when I was CTO of State Software. In 2002 I acquired the `json.org` domain and put up a page describing the format. With no other effort on my part, JSON has been widely adopted by people who found that it made it a lot easier to produce distributed applications and services. The original page has been translated into Chinese, French, German, Italian, Japanese, Korean, and Spanish. JSON has been formalized in RFC 4627. The MIME Media Type is `application/json`.

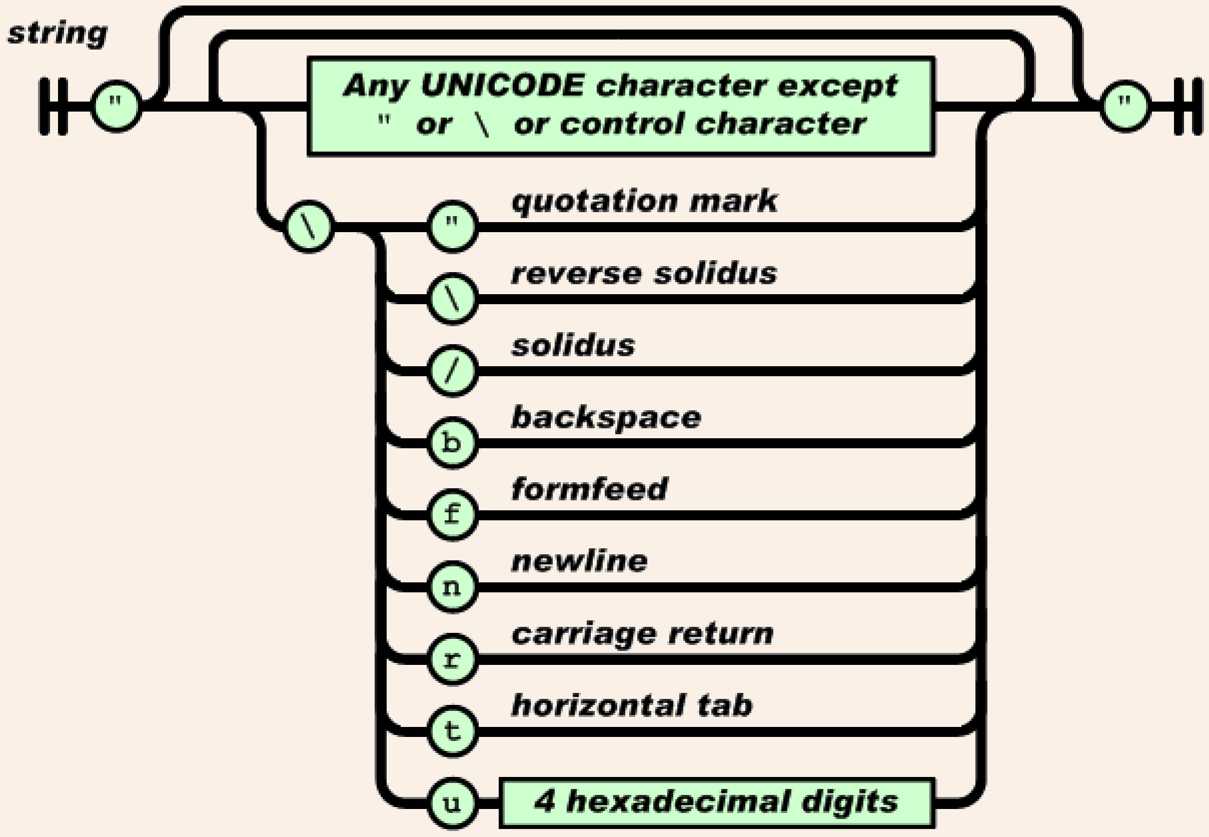
There are now JSON libraries or built-in JSON support for these programming languages and systems: ActionScript, C, C++, C#, Cold Fusion, D, Delphi, E, Erlang, Haskell, Java, Lisp, LotusScript, Lua, Perl, Objective-C, OCAML, PHP, Python, Rebol, Ruby, Scheme, and Squeak.

JSON is based on the object quasi-literals of JavaScript. Coincidentally, Python has the same notation, as does Newtonscript (which John Scully said will be the basis of a trillion dollar industry). There are many other data formats which are quite similar to JSON. JSON is a natural representation of data for the C family of programming languages.

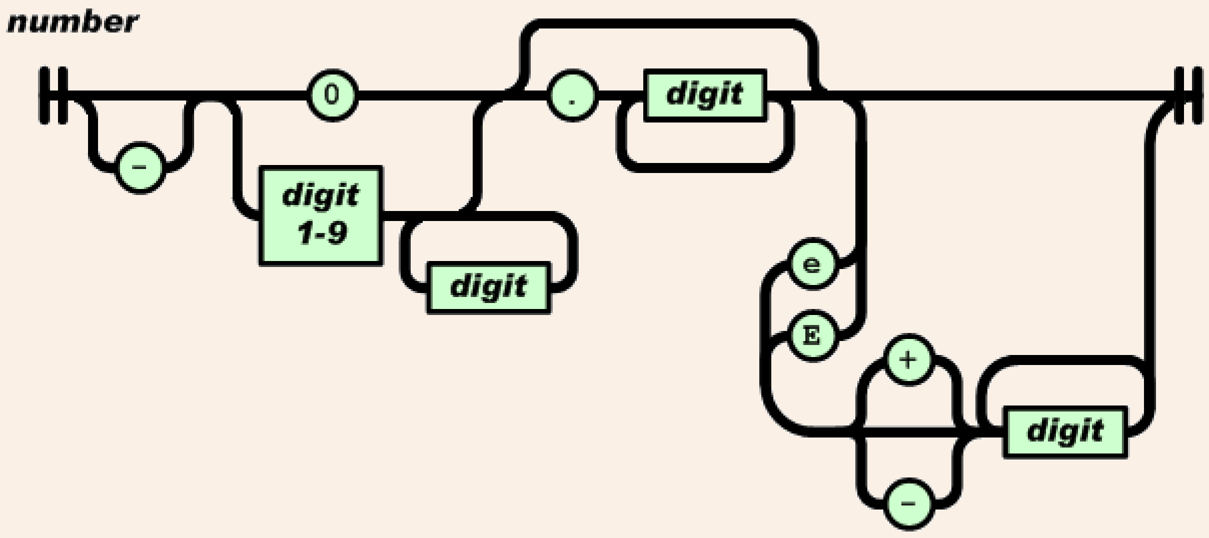
The types represented in JSON are strings, numbers, booleans, object, arrays, and `null`. JSON syntax is nicely expressed in railroad diagrams.



A string is a sequence of zero or more characters wrapped in quotes with backslash escapement, the same notation used in most programming languages.

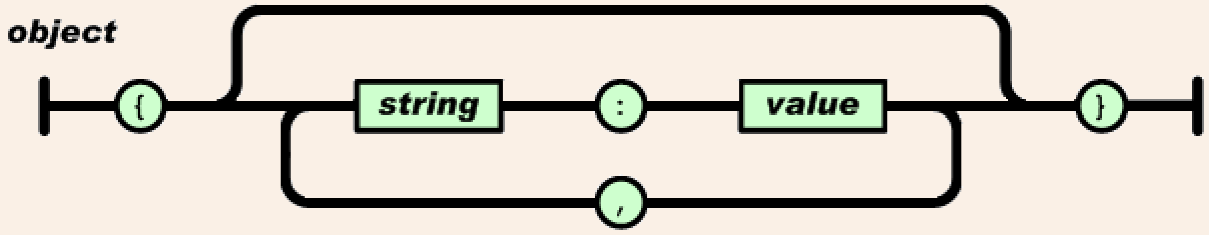


A number can be represented as integer, real, or floating point. JSON does not support octal or hex because it is minimal. It does not have values for `NaN` or `Infinity` because it does not want to be tied to any particular internal representation. Numbers are not quoted. It would be insane to require quotes around numbers.



You'll be glad to know that JSON has exactly two boolean values, and that they are `true` and `false`. If it had exactly zero boolean values, it wouldn't be a data format.

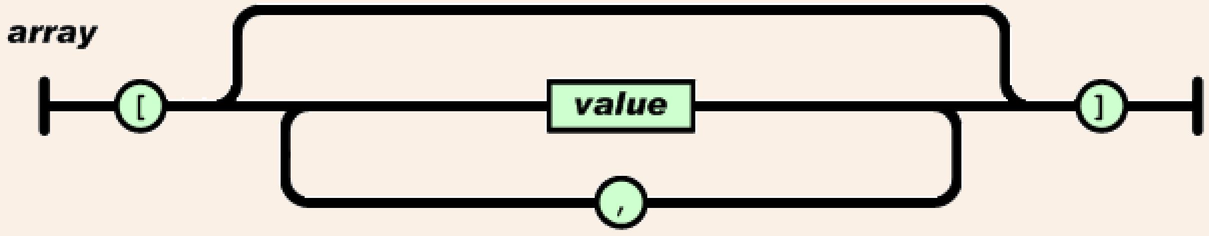
A JSON object is an unordered collection of key/value pairs. The keys are strings and the values are any of the JSON types. A colon separates the keys from the values, and comma separates the pairs. The whole thing is wrapped in curly braces. It maps directly onto objects, structs, records, and hashtables.



This is an example of a JSON object.

```
{
    "name": "Jack (\"Bee\") Nimble",
    "format": {
        "type":       "rect",
        "width":      1920,
        "height":     1080,
        "interlace":  false,
        "frame rate": 24
    }
}
```

The JSON array is an ordered collection of values separated by commas. The whole thing is wrapped in square brackets. It maps directly onto arrays, vectors, and lists.



These are examples of JSON arrays:

```
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```

```
[
    [0, -1, 0],
    [1, 0, 0],
    [0, 0, 1]
]
```

The character encoding of JSON text is always Unicode. UTF-8 is the only encoding that makes sense on the wire, but UTF-16 and UTF-32 are also permitted.

JSON has no version number. No revisions to the JSON grammar are anticipated. If something has a 1.0, it will inevitably get a 1.1 and a 2.0, and everything is crap until it is 3.0. JSON is very stable.

A JSON decoder MUST accept all well-formed JSON text. A JSON decoder MAY also accept non-JSON text. A JSON encoder MUST only produce well-formed JSON text. This is consistent with Postel's Law: "Be liberal in what you accept, and conservative in what you send."

This allows JSON supersets. JavaScript is a JSON superset. YAML is also a JSON superset. JSON may become the common core of an interesting class of languages and formats.

JSON has become the X in Ajax. It is now the preferred data format for Ajax applications. There are a number of ways in which JSON can be used in Ajax applications. The first is to include JSON text in the original HTML.

```
<html>...
<script>
var data = JSONdata;
</script>...
</html>
```

This is useful in cases where the JSON text is significantly smaller than its HTML representation. By completing the HTML generation in JavaScript, the page can be delivered more quickly.

The most common way to use JSON is with XMLHttpRequest. Once a response text obtained, it can quickly be converted into a JavaScript data structure and consumed by the program. There are two ways in which to do the conversion. The first is to use JavaScript's `eval` function, which will invoke the JavaScript compiler.

```
responseData = eval('(' + responseText + ')');
```

This works because JSON is a safe subset of JavaScript, but it is potentially dangerous because whatever the server sends will be executed. XMLHttpRequest is severely limited by the same origin policy, so the response text can only come from the originining server. If the server acts as a proxy and is incompetent in its filtering, then it could include dangerous scripts in the response text. If there is any risk of this, then the `parseJSON` method must be used instead.

```
responseData = responseText.parseJSON();
```

The `parseJSON` method will be included in the Fourth Edition of ECMAScript. In the meantime, a JavaScript implementation is available at `json.org`.

Another approach is to use an invisible `<iframe>` for data communication. This allows for partial circumvention of the same origin policy in that communication with a different subdomain is possible. The server sends JSON text embedded in a script in a document.

```
<html><head><script>
   document.domain = 'penzance.com';
   parent.deliver(JSONtext);
</script></head></html>
```

The function `deliver` is passed the incoming data structure.

A popular alternative is use of the dynamic script tag hack. It completely circumvents the same origin policy so that data can be obtained from any server in the world. It is much easier to use than XMLHttpRequest. Just create a script node. The server sends the JSON text embedded in a script.

```
deliver(JSONtext);
```

The function `deliver` is passed the incoming data structure. There is no opportunity to inspect the response before it is evaluated, so there is no defense against a malevolent server sending a dangerous script instead of JSON text. The dynamic script tag hack is insecure. It should not be used.

I am recommending a new data communication facility that will permit safe two-way data interchange between any page and any server. It will be exempt from the Same Origin Policy and introduce no new security vulnerabilities. Tell your favorite browser maker "I want JSONRequest!" You can read more about it at `http://www.JSON.org/JSONRequest.html`.

Now that your JavaScript program has received the data, what can it do with it? One of the simplest things is client-side HTML generation.

```
var template = '<table border="{border}"><tr><th>Last</th><td>{last}</td></tr>' +
   '<tr><th>First</th><td>{first}</td></tr></table>';
```

Notice that we have an HTML template with three variables in it. Then we'll obtain a JSON object containing members that match the variables.

```
var data = {
    "first": "Carl",
    "last": "Hollywood",
    "border": 2
};
```

We can then use a `supplant` method to fill in the template with the data.

```
mydiv.innerHTML = template.supplant(data);
```

JavaScript strings do not come with a `supplant` method, but that is ok because JavaScript allows us to augment the built-in types, giving them the features we need.

```
String.prototype.supplant = function (o) {
    return this.replace(/{([^{}]*)}/g,
        function (a, b) {
            var r = o[b];
            return typeof r === 'string' ?
                r : a;
        }
    );
};
```

A much more interesting class of transformations is permitted by the JSONT package.

For example, we invoke jsonT with an object containing a set of rules (where the rules can be templates, patterns, or JavaScript functions) and a data object with a complicated structure.

```
var rules = {
   self: '<svg><{closed} stroke="{color}" points="{points}" /></svg>',
   closed: function (x) {return x ? 'polygon' : 'polyline';},
   'points[*][*]': '{$} '
   };

var data = {
   "color": "blue",
   "closed": true,
   "points": [[10,10], [20,10], [20,20], [10,20]]
   };

jsonT(data, rules)

<svg><polygon stroke="blue" points="10 10 20 10 20 20 10 20 " /></svg>
```

JavaScript is such a powerful language that the JSONT transformer is less than a page of code. See `http://goessner.net/articles/jsont/`.

The characteristics of XML that make it suitable for data interchange are stronger in JSON.

> Its simultaneously human- and machine-readable format;

This is true of both formats.

> It has support for Unicode, allowing almost any information in any human language to be communicated;

JSON uses Unicode exclusively.

> The self-documenting format that describes structure and field names as well as specific values;

This is also true of both formats, and it raises the question: If the format is self-describing, why is it necessary to have a schema?

> The strict syntax and parsing requirements that allow the necessary parsing algorithms to remain simple, efficient, and consistent;

JSON's syntax is significantly simpler, so parsing is more efficient.

> The ability to represent the most general computer science data structures: records, lists and trees.

This is the most significant difference. While there are transformations which allow XML to express these structures, JSON expresses them directly. JSON's simple values are the same as used in programming languages. JSON's structures look like conventional programming language structures. No restructuring is necessary. JSON's object is record, struct, object, dictionary, hash, or associative array. JSON's array is array, vector, sequence, or list.

JSON doesn't have namespaces. Every object is a namespace: its set of keys is independent of all other objects, even exclusive of nesting. JSON uses context to avoid ambiguity, just as programming languages do.

JSON has no validator. Being well-formed and valid is not the same as being correct and relevant. Ultimately, every application is responsible for validating its inputs. This cannot be delegated. YAML has a validator, and since YAML is a superset of JSON, that validator can be used.

JSON is not extensible. It does not need to be. It can represent any non-recurrent data structure as is. JSON is flexible. New fields can be added to existing structures without obsoleting existing programs. In this sense, XML is also not extensible. It is possible to add new tags and attributes, but it is not possible to extend XML to add expressive syntax for arrays and objects and numbers and booleans.