JSONPath WG
Internet-Draft
Intended status: Standards Track
Expires: 10 June 2021

G. Normington, Ed.
VMware, Inc.
E. Surov, Ed.
TheSoul Publishing Ltd.
M. Mikulicic
VMware, Inc.
S. Gössner
Fachhochschule Dortmund
7 December 2020

JavaScript Object Notation (JSON) Path
draft-normington-jsonpath-00

Abstract

JSONPath defines a string syntax for identifying values within a JavaScript Object Notation (JSON) document.

Note

This document is a work in progress and has not yet been published as an Internet Draft (which needs to be fixed soon).

Contributing

This document picks up the popular JSONPath specification dated 2007-02-21 and provides a normative definition for it. In its current state, it is a strawman document showing what needs to be covered.

Comments and issues can be directed at the github repository _insert repo here_ as well as (for the time when the more permanent home is being decided) at the dispatch@ietf.org mailing list.

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 https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Introduction

This document picks up the popular JSONPath specification dated 2007-02-21 [JSONPath-orig] and provides a normative definition for it. In its current state, it is a strawman document showing what needs to be covered.

JSON is defined by [RFC8259].

JSONPath is not intended as a replacement, but as a more powerful companion, to JSON Pointer [RFC6901]. [insert reference to section where the relationship is detailed. The purposes of the two syntaxes are different. Pointer is for isolating a single location within a document. Path is a query syntax that can also be used to pull multiple locations.]

1.1. Terminology

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 ABNF, as described in [RFC5234]. ABNF terminal values in this document define Unicode code points rather than their UTF-8 encoding. For example, the Unicode PLACE OF INTEREST SIGN (U+2318) would be defined in ABNF as "%x2318".

The terminology of [RFC8259] applies.

Data Item: A structure complying to the generic data model of JSON, i.e., composed of containers such as arrays and maps (JSON objects), and of atomic data such as null, true, false, numbers, and text strings.

Object: Used in its generic sense, e.g., for programming language objects. When a JSON Object as defined in [RFC8259] is meant, we specifically say JSON Object.

Query: Short name for JSONPath expression.

Argument: Short name for the JSON data item a JSONPath expression is applied to.

Output Path: A simple form of JSONPath expression that identifies a Position by providing a query that results in exactly that position. Similar to, but syntactically different from, a JSON Pointer [RFC6901].

Position: A JSON data item identical to or nested within the JSON data item to which the query is applied to, expressed either by the value of that data item or by providing a JSONPath Output Path.

1.2. Inspired by XPath

A frequently emphasized advantage of XML is the availability of powerful tools to analyse, transform and selectively extract data from XML documents. [XPath] is one of these tools.

In 2007, the need for something solving the same class of problems for the emerging JSON community became apparent, specifically for:

* Finding data interactively and extracting them out of [RFC8259] data items without special scripting.

* Specifying the relevant parts of the JSON data in a request by a client, so the server can reduce the amount of data in its response, minimizing bandwidth usage.

So what does such a tool look like for JSON? When defining a JSONPath, how should expressions look?

The XPath expression

/store/book[1]/title

looks like

x.store.book[0].title

or

x['store']['book'][0]['title']

in popular programming languages such as JavaScript, Python and PHP, with a variable x holding the JSON data item. Here we observe that such languages already have a fundamentally XPath-like feature built in.

The JSONPath tool in question should:

* be naturally based on those language characteristics.

* cover only essential parts of XPath 1.0.

* be lightweight in code size and memory consumption.

* be runtime efficient.

1.3. Overview of JSONPath Expressions

JSONPath expressions always apply to a JSON data item in the same way as XPath expressions are used in combination with an XML document. Since a JSON data item is usually anonymous and doesn't necessarily have a "root member object", JSONPath used the abstract name "$" to refer to the top level object of the data item.

JSONPath expressions can use the _dot-notation_

$.store.book[0].title

or the _bracket-notation_

$['store']['book'][0]['title']

for paths input to a JSONPath processor. [1] Where a JSONPath processor uses JSONPath expressions as output paths, these will always be converted to the more general _bracket-notation_. [2] Bracket notation is more general than dot notation and can serve as a canonical form when a JSONPath processor uses JSONPath expressions as output paths.

JSONPath allows the wildcard symbol "*" for member names and array indices. It borrows the descendant operator ".." from [E4X] and the array slice syntax proposal "[start:end:step]" [SLICE] from ECMASCRIPT 4.

JSONPath was originally designed to employ an _underlying scripting language_ for computing expressions. The present specification defines a simple expression language that is independent from any scripting language in use on the platform.

JSONPath can use expressions, written in parentheses: "(<expr>)", as an alternative to explicit names or indices as in:

$.store.book[(@.length-1)].title

The symbol "@" is used for the current object. Filter expressions are supported via the syntax "?(<boolean expr>)" as in

```
$.store.book[?(@.price < 10)].title
```

Here is a complete overview and a side by side comparison of the JSONPath syntax elements with their XPath counterparts.

XPath	JSONPath	Description
/	$	the root object/element
.	@	the current object/element
/	"." or "[]"	child operator
..	n/a	parent operator
//	..	nested descendants (JSONPath borrows this syntax from E4X)
*	*	wildcard: All objects/elements regardless of their names
@	n/a	attribute access: JSON data items do not have attributes
[]	[]	subscript operator: XPath uses it to iterate over element collections and for predicates; native array indexing as in JavaScript here
|	[,]	Union operator in XPath (results in a combination of node sets); JSONPath allows alternate names or array indices as a set
n/a	[start:end:step]	array slice operator borrowed from ES4
[]	?()	applies a filter (script) expression
n/a	()	expression engine
()	n/a	grouping in Xpath

Table 1: Overview over JSONPath, comparing to XPath

XPath has a lot more to offer (location paths in unabbreviated syntax, operators and functions) than listed here. Moreover there is a significant difference how the subscript operator works in Xpath and JSONPath:

* Square brackets in XPath expressions always operate on the _node set_ resulting from the previous path fragment. Indices always start at 1.

* With JSONPath, square brackets operate on the _object_ or _array_ addressed by the previous path fragment. Array indices always start at 0.

2. JSONPath Examples

This section provides some more examples for JSONPath expressions. The examples are based on a simple JSON data item patterned after a typical XML example representing a bookstore (that also has bicycles):

```
{ "store": {
    "book": [
      { "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      { "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      { "category": "fiction",
        "author": "Herman Melville",
        "title": "Moby Dick",
        "isbn": "0-553-21311-3",
        "price": 8.99
      },
      { "category": "fiction",
        "author": "J. R. R. Tolkien",
        "title": "The Lord of the Rings",
        "isbn": "0-395-19395-8",
        "price": 22.99
      }
    ],
    "bicycle": {
      "color": "red",
      "price": 19.95
    }
  }
}
```

Figure 1: Example JSON data item

The examples in Table 2 use the expression mechanism to obtain the number of items in an array, to test for the presence of a map member, and to perform numeric comparisons of map member values with a constant.

XPath	JSONPath	Result
/store/book/author	$.store.book[*].author	the authors of all books in the store
//author	$..author	all authors
/store/*	$.store.*	all things in store, which are some books and a red bicycle
/store//price	$.store..price	the prices of everything in the store
//book[3]	$..book[2]	the third book
//book[last()]	"$..book[(@.length-1)]" "$..book[-1]"	the last book in order
//book[position()<3]	"$..book[0,1]" "$..book[:2]"	the first two books
//book[isbn]	$..book[?(@.isbn)]	filter all books with isbn number
//book[price<10]	$..book[?(@.price<10)]	filter all books cheaper than 10
//*	$..*	all elements in XML document; all members of JSON data item

Table 2: Example JSONPath expressions applied to the example JSON data item

3. JSONPath Syntax and Semantics

3.1. Overview

A JSONPath is a string which selects zero or more nodes of a piece of JSON. A valid JSONPath conforms to the ABNF syntax defined by this document.

A JSONPath MUST be encoded using UTF-8. To parse a JSONPath according to the grammar in this document, its UTF-8 form SHOULD first be decoded into Unicode code points as described in [RFC3629].

3.2. Terminology

A JSON value is logically a tree of nodes.

Each node holds a JSON value (as defined by [RFC8259]) of one of the types object, array, number, string, or one of the literals "true", "false", or "null". The type of the JSON value held by a node is sometimes referred to as the type of the node.

3.3. Implementation

An implementation of this specification, from now on referred to simply as "an implementation", SHOULD takes two inputs, a JSONPath and a JSON value, and produce a possibly empty list of nodes of the JSON value which are selected by the JSONPath or an error (but not both).

If no node is selected and no error has occurred, an implementation MUST return an empty list of nodes.

Syntax errors in the JSONPath SHOULD be detected before selection is attempted since these errors do not depend on the JSON value. Therefore, an implementation SHOULD take a JSONPath and produce an optional syntax error and then, if and only if an error was not produced, SHOULD take a JSON value and produce a list of nodes or an error (but not both).

Alternatively, an implementation MAY take a JSONPath and a JSON value and produce a list of nodes or an optional error (but not both).

For any implementation, if a syntactically invalid JSONPath is provided, the implementation MUST return an error.

If a syntactially invalid JSON value is provided, any implementation SHOULD return an error.

3.4. Syntax

Syntactically, a JSONPath consists of a root selector ("$"), which selects the root node of a JSON value, followed by a possibly empty sequence of _selectors_.

```
json-path = root-selector *selector
root-selector = %x24              ; $ selects document root node
```

The syntax and semantics of each selector is defined below.

3.5. Semantics

The root selector "$" not only selects the root node of the input document, but it also produces as output a list consisting of one node: the input document.

A selector may select zero or more nodes for further processing. A syntactically valid selector MUST NOT produce errors. This means that some operations which might be considered erroneous, such as indexing beyond the end of an array, simply result in fewer nodes being selected.

But a selector doesn't just act on a single node: each selector acts on a list of nodes and produces a list of nodes, as follows.

After the root selector, the remainder of the JSONPath is processed by passing lists of nodes from one selector to the next ending up with a list of nodes which is the result of applying the JSONPath to the input JSON value.

Each selector acts on its input list of nodes as follows. For each node in the list, the selector selects zero or more nodes, each of which is a descendant of the node or the node itself. The output list of nodes of a selector is the concatenation of the lists of selected nodes for each input node.

A specific, non-normative example will make this clearer. Suppose the input document is: "{"a":[{"b":0},{"b":1},{"c":2}]}". As we will see later, the JSONPath "$.a[*].b" selects the following list of nodes: "0", "1". Let's walk through this in detail.

The JSONPath consists of "$" followed by three selectors: ".a", "[*]", and ".b".

Firstly, "$" selects the root node which is the input document. So the result is a list consisting of just the root node.

Next, ".a" selects from any input node of type object and selects any value of the input node corresponding to the key ""a"". The result is again a list of one node: "[{"b":0},{"b":1},{"c":2}]".

Next, "[*]" selects from any input node which is an array and selects all the elements of the input node. The result is a list of three nodes: "{"b":0}", "{"b":1}", and "{"c":2}".

Finally, ".b" selects from any input node of type object with a key "b" and selects the value of the input node corresponding to that key. The result is a list containing "0", "1". This is the concatenation of three lists, two of length one containing "0", "1", respectively, and one of length zero.

As a consequence of this approach, if any of the selectors selects no nodes, then the whole JSONPath selects no nodes.

In what follows, the semantics of each selector are defined for each type of node.

3.6. Selectors

3.6.1. Dot Child Selector

Syntax

A dot child selector has a key known as a dot child name or a single asterisk ("*").

A dot child name corresponds to a name in a JSON object.

```
selector = dot-child              ; see below for alternatives
dot-child = %x2E dot-child-name / ; .<dot-child-name>
            %x2E %x2A             ; .*
dot-child-name = 1*(
                   %x2D /         ; -
                   DIGIT /
                   ALPHA /
                   %x5F /         ; _
                   %x80-10FFFF    ; any non-ASCII Unicode character
                 )
DIGIT =  %x30-39                  ; 0-9
ALPHA = %x41-5A / %x61-7A         ; A-Z / a-z
```

More general child names, such as the empty string, are supported by "Union Child" (Section 3.6.2.3).

Note that the "dot-child-name" rule follows the philosophy of JSON strings and is allowed to contain bit sequences that cannot encode Unicode characters (a single unpaired UTF-16 surrogate, for example). The behaviour of an implementation is undefined for child names which do not encode Unicode characters.

Semantics

A dot child name which is not a single asterisk ("*") is considered to have a key. It selects the value corresponding to the key from any object node. It selects no nodes from a node which is not an object.

The key of a dot child name is the sequence of Unicode characters contained in that name.

A dot child name consisting of a single asterisk is a wild card. It selects all the values of any object node. It also selects all the elements of any array node. It selects no nodes from number, string, or literal nodes.

3.6.2. Union Selector

3.6.2.1. Syntax

A union selector consists of one or more union elements.

```
selector =/ union
union = %x5B ws union-elements ws %x5D ; [...]
ws = *%x20                             ; zero or more spaces
union-elements = union-element /
                 union-element ws %x2C ws union-elements
                                       ; ,-separated list
```

3.6.2.2. Semantics

A union selects any node which is selected by at least one of the union selectors and selects the concatenation of the lists (in the order of the selectors) of nodes selected by the union elements.

3.6.2.3. Child

Syntax

A child is a quoted string.

```
union-element = child ; see below for more alternatives
child = %x22 *double-quoted %x22 / ; "string"
        %x27 *single-quoted %x27   ; 'string'

double-quoted = dq-unescaped /
          escape (
              %x22 /          ; "    quotation mark  U+0022
              %x2F /          ; /    solidus         U+002F
              %x5C /          ; \    reverse solidus U+005C
              %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


      dq-unescaped = %x20-21 / %x23-5B / %x5D-10FFFF

single-quoted = sq-unescaped /
          escape (
              %x27 /          ; '    apostrophe      U+0027
              %x2F /          ; /    solidus         U+002F
              %x5C /          ; \    reverse solidus U+005C
              %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

      sq-unescaped = %x20-26 / %x28-5B / %x5D-10FFFF

escape = %x5C                 ; \

HEXDIG =  DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
                              ; case insensitive hex digit
```

Notes: 1. double-quoted strings follow JSON in [RFC8259]. Single-quoted strings follow an analogous pattern. 2. "HEXDIG" includes A-F and a-f.

Semantics

If the child is a quoted string, the string MUST be converted to a key by removing the surrounding quotes and replacing each escape sequence with its equivalent Unicode character, as in the table below:

Escape Sequence	Unicode Character
%x5C %x22	U+0022
%x5C %x27	U+0027
%x5C %x2F	U+002F
%x5C %x5C	U+005C
%x5C %x62	U+0008
%x5C %x66	U+000C
%x5C %x6E	U+000A
%x5C %x72	U+000D
%x5C %x74	U+0009
%x5C uXXXX	U+XXXX

Table 3: Escape Sequence Replacements

The child selects the value corresponding to the key from any object node with the key as a name. It selects no nodes from a node which is not an object.

3.6.2.4. Array Selector

Syntax

An array selector selects zero or more elements of an array node. An array selector takes the form of an index, which selects at most one element, or a slice, which selects zero or more elements.

```
union-element =/ array-index / array-slice
```

An array index is an integer (in base 10).

```
array-index = integer

integer = ["-"] ("0" / (DIGIT1 *DIGIT))
                            ; optional - followed by 0 or
                            ; sequence of digits with no leading zero
DIGIT1 = %x31-39            ; non-zero digit
```

Note: the syntax does not allow integers with leading zeros such as "01" and "-01".

An array slice consists of three optional integers (in base 10) separated by colons.

```
array-slice = [ start ] ws ":" ws [ end ]
                    [ ws ":" ws [ step ] ]
start = integer
end = integer
step = integer
```

Note: the array slices ":" and "::" are both syntactically valid, as are ":2:2", "2::2", and "2:4:".

Semantics

Informal Introduction

This section is non-normative.

Array indexing is a way of selecting a particular element of an array using a 0-based index. For example, the expression "[0]" selects the first element of a non-empty array.

Negative indices index from the end of an array. For example, the expression "[-2]" selects the last but one element of an array with at least two elements.

Array slicing is inspired by the behaviour of the "Array.prototype.slice" method of the JavaScript language as defined by the ECMA-262 standard [ECMA-262], with the addition of the "step" parameter, which is inspired by the Python slice expression.

The array slice expression "[start:end:step]" selects elements at indices starting at "start", incrementing by "step", and ending with "end" (which is itself excluded). So, for example, the expression "[1:3]" (where "step" defaults to "1") selects elements with indices "1" and "2" (in that order) whereas "[1:5:2]" selects elements with indices "1" and "3".

When "step" is negative, elements are selected in reverse order. Thus, for example, "[5:1:-2]" selects elements with indices "5" and "3", in that order and "[::-1]" selects all the elements of an array in reverse order.

When "step" is "0", no elements are selected. This is the one case which differs from the behaviour of Python, which raises an error in this case.

The following section specifies the behaviour fully, without depending on JavaScript or Python behaviour.

Detailed Semantics

An array selector is either an array slice or an array index, which is defined in terms of an array slice.

A slice expression selects a subset of the elements of the input array, in the same order as the array or the reverse order, depending on the sign of the "step" parameter. It selects no nodes from a node which is not an array.

A slice is defined by the two slice parameters, "start" and "end", and an iteration delta, "step". Each of these parameters is optional. "len" is the length of the input array.

The default value for "step" is "1". The default values for "start" and "end" depend on the sign of "step", as follows:

Condition	start	end
step >= 0	0	len
step < 0	len - 1	-len - 1

Table 4: Default array slice start and end values

Slice expression parameters "start" and "end" are not directly usable as slice bounds and must first be normalized. Normalization is defined as:

```
FUNCTION Normalize(i):
  IF i >= 0 THEN
    RETURN i
  ELSE
    RETURN len + i
  END IF
```

The result of the array indexing expression "[i]" is defined to be the result of the array slicing expression "[i:Normalize(i)+1:1]".

Slice expression parameters "start" and "end" are used to derive slice bounds "lower" and "upper". The direction of the iteration, defined by the sign of "step", determines which of the parameters is the lower bound and which is the upper bound:

```
FUNCTION Bounds(start, end, step, len):
  n_start = Normalize(start)
  n_end = Normalize(end)

  IF step >= 0 THEN
    lower = MIN(MAX(n_start, 0), len)
    upper = MIN(MAX(n_end, 0), len)
  ELSE
    upper = MIN(MAX(n_start, -1), len-1)
    lower = MIN(MAX(n_end, -1), len-1)
  END IF

  RETURN (lower, upper)
```

The slice expression selects elements with indices between the lower and upper bounds. In the following pseudocode, the "a(i)" construct expresses the 0-based indexing operation on the underlying array.

```
IF step > 0 THEN

  i = lower
  WHILE i < upper:
    SELECT a(i)
    i = i + step
  END WHILE

ELSE if step < 0 THEN

  i = upper
  WHILE lower < i:
    SELECT a(i)
    i = i + step
  END WHILE

END IF
```

When "step = 0", no elements are selected and the result array is empty.

An implementation MUST raise an error if any of the slice expression parameters does not fit in the implementation's representation of an integer. If a successfully parsed slice expression is evaluated against an array whose size doesn't fit in the implementation's representation of an integer, the implementation MUST raise an error.

4. IANA Considerations

TBD: Define a media type for JSON Path expressions.

5. Security Considerations

This section gives security considerations, as required by [RFC3552].

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 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", STD 68, RFC 5234, DOI 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", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

6.2. Informative References

[E4X] ISO, "Information technology ECMAScript for XML (E4X) specification", ISO/IEC 22537:2006 , 2006.

[ECMA-262] Ecma International, "ECMAScript Language Specification, Standard ECMA-262, Third Edition", December 1999, <http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf>.

[JSONPath-orig] Gössner, S., "JSONPath XPath for JSON", 21 February 2007, <https://goessner.net/articles/JsonPath/>.

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/info/rfc6901>.

[SLICE] "Slice notation", n.d., <https://github.com/tc39/proposal-slice-notation>.

[XPath] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and J. Simeon, "XML Path Language (XPath) 2.0 (Second Edition)", World Wide Web Consortium Recommendation REC-xpath20-20101214, 14 December 2010, <https://www.w3.org/TR/2010/REC-xpath20-20101214>.

Acknowledgements

This specification is based on Stefan Gössner's original online article defining JSONPath [JSONPath-orig].

The books example was taken from http://coli.lili.uni-bielefeld.de/~andreas/Seminare/sommer02/books.xml -- a dead link now.

Contributors

Carsten Bormann
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

Phone: +49-421-218-63921
Email: cabo@tzi.org

Authors' Addresses

Glyn Normington (editor)
VMware, Inc.
Winchester
United Kingdom

Email: glyn.normington@gmail.com

Edward Surov (editor)
TheSoul Publishing Ltd.
Limassol
Cyprus

Email: esurov.tsp@gmail.com

Marko Mikulicic
VMware, Inc.
Pisa
Italy

Email: mmikulicic@gmail.com

Stefan Gössner
Fachhochschule Dortmund
Sonnenstraße 96
D-44139 Dortmund
Germany

Email: stefan.goessner@fh-dortmund.de