Workgroup: JSONPath WG
Internet-Draft: draft-ietf-jsonpath-base-02
Published: 25 October 2021
Intended Status: Standards Track
Expires: 28 April 2022
Authors: S. Gössner, Ed. G. Normington, Ed.
Fachhochschule Dortmund
C. Bormann, Ed.
Universität Bremen TZI

JSONPath: Query expressions for JSON

Abstract

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

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 may be directed to this document's github repository.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction
  1.1. Terminology
  1.2. Inspired by XPath
  1.3. Overview of JSONPath Expressions
2. JSONPath Examples
3. JSONPath Syntax and Semantics
  3.1. Overview
  3.2. Processing Model
  3.3. Syntax
  3.4. Semantics
  3.5. Selectors
    3.5.1. Root Selector
    3.5.2. Dot Selector
    3.5.3. Dot Wild Card Selector
    3.5.4. Index Selector
    3.5.5. Index Wild Card Selector
    3.5.6. Array Slice Selector
    3.5.7. Descendant Selector
    3.5.8. Union Selector
      3.5.8.1. Syntax
      3.5.8.2. Semantics
    3.5.9. Filter Selector
      3.5.9.1. Syntax
      3.5.9.2. Semantics
4. Expression Language
5. IANA Considerations
6. Security Considerations
7. References
  7.1. Normative References
  7.2. Informative References
Acknowledgements
Contributors
Authors' Addresses

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 except where clarified below. The terms "Primitive" and "Structured" are used to group the types as in Section 1 of [RFC8259]. Definitions for "Object", "Array", "Number", and "String" remain unchanged. Importantly "object" and "array" in particular do not take on a generic meaning, such as they would in a general programming context.

Additional terms used in this specification are defined below.

Value: As per [RFC8259], a structure complying to the generic data model of JSON, i.e., composed of components such as containers, namely JSON objects and arrays, and atomic data, namely null, true, false, numbers, and text strings.

Member: A name/value pair in an object. (Not itself a value.)

Name: The name in a name/value pair constituting a member. (Also known as "key", "tag", or "label".) This is also used in [RFC8259], but that specification does not formally define it. It is included here for completeness.

Element:
A value in an array. (Also used with a distinct meaning in XML context for XML elements.)

Index: A non-negative integer that identifies a specific element in an array.

Query: Short name for JSONPath expression.

Argument: Short name for the value a JSONPath expression is applied to.

Node: The pair of a value along with its location within the argument.

Root Node: The unique node whose value is the entire argument.

Nodelist: A list of nodes. The output of applying a query to an argument is manifested as a list of nodes. While this list can be represented in JSON, e.g. as an array, the nodelist is an abstract concept unrelated to JSON values.

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

For the purposes of this specification, a value as defined by [RFC8259] is also viewed as a tree of nodes. Each node, in turn, holds a value. Further nodes within each value are the elements of arrays and the member values of objects and are themselves values. (The type of the value held by a node may also be referred to as the type of the node.)

A query is applied to an argument, and the output is a nodelist.

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] JSON values 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 argument. 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 value in the same way as XPath expressions are used in combination with an XML document. Since a value is anonymous, JSONPath uses the abstract name $ to refer to the root node of the argument.

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 Output Paths which employ 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 node. 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 element/node
.	@	the current element/node
/	. or []	child operator
..	n/a	parent operator
//	..	nested descendants (JSONPath borrows this syntax from E4X)
*	*	wildcard: All elements/nodes regardless of their names
@	n/a	attribute access: JSON values 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 the simple JSON value shown in Figure 1, which was 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 value

The examples in Table 2 use the expression mechanism to obtain the number of elements in an array, to test for the presence of a member in a object, and to perform numeric comparisons of 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

XPath	JSONPath	Result
//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 member values and array elements contained in input value

Table 2: Example JSONPath expressions applied to the example JSON value

3. JSONPath Syntax and Semantics

3.1. Overview

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

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

A string to be used as a JSONPath query needs to be *well-formed* and *valid*. A string is a well-formed JSONPath query if it conforms to the syntax of JSONPath. A well-formed JSONPath query is valid if it also fulfills all semantic requirements posed by this document.

The well-formedness and the validity of JSONPath queries are independent of the value the query is applied to; no further errors can be raised during application of the query to a value.

(Obviously, an implementation can still fail when executing a JSONPath query, e.g., because of resource depletion, but this is not modeled in the present specification.)

3.2. Processing Model

In this specification, the semantics of a JSONPath query are defined in terms of a *processing model*. That model is not prescriptive of the internal workings of an implementation: Implementations may wish (or need) to design a different process that yields results that conform to the model.

In the processing model, a valid query is executed against a value, the *argument*, and produces a list of zero or more nodes of the value.

The query is a sequence of zero or more *selectors*, each of which is applied to the result of the previous selector and provides input to the next selector. These results and inputs take the form of a *nodelist*, i.e., a sequence of zero or more nodes.

The nodelist going into the first selector contains a single node, the argument. The nodelist resulting from the last selector is presented as the result of the query; depending on the specific API, it might be presented as an array of the JSON values at the nodes, an array of Output Paths referencing the nodes, or both -- or some other representation as desired by the implementation. Note that the API must be capable of presenting an empty nodelist as the result of the query.

A selector performs its function on each of the nodes in its input nodelist, during such a function execution, such a node is referred to as the "current node". Each of these function executions produces a nodelist, which are then concatenated into the result of the selector.

The processing within a selector may execute nested queries, which are in turn handled with the processing model defined here. Typically, the argument to that query will be the current node of the selector or a set of nodes subordinate to that current node.

3.3. Syntax

Syntactically, a JSONPath query consists of a root selector ($), which stands for a nodelist that contains the root node of the argument, followed by a possibly empty sequence of *selectors*.

```
json-path = root-selector *(dot-selector        /
                            dot-wild-selector   /
                            index-selector      /
                            index-wild-selector /
                            union-selector      /
                            slice-selector      /
                            descendant-selector /
                            filter-selector)
```

The syntax and semantics of each selector is defined below.

3.4. Semantics

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

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: a selector acts on each of the nodes in its input nodelist and concatenates the resultant nodelists to form the result nodelist of the selector.

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.

For instance, with the argument {"a":[{"b":0},{"b":1},{"c":2}]}, the query $.a[*].b selects the following list of nodes: 0, 1 (denoted here by their value). Let's walk through this in detail.

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

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

Next, .a selects from any input node of type object and selects the node of any member value of the input node corresponding to the member name "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 member name b and selects the node of the member value of the input node corresponding to that name. 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 query selects no nodes.

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

3.5. Selectors

A JSONPath query consists of a sequence of selectors. Valid selectors are

- *Root selector $
- *Dot selector .<name>, used with object member names exclusively.
- *Dot wild card selector .*.
- *Index selector [<index>], where <index> is either an (possibly negative) array index or an object member name.
- *Index wild card selector [*].
- *Array slice selector [<start>:<end>:<step>], where <start>, <end>, <step> are integer literals.
- *Nested descendants selector ...
- *Union selector [<sel1>,<sel2>,...,<selN>], holding a comma delimited list of index, index wild card, array slice, and filter selectors.
- *Filter selector [?(<expr>)]
- *Current item selector @

3.5.1. Root Selector

Syntax

Every valid JSONPath query MUST begin with the root selector $.

```
root-selector  = "$"
```

Semantics

The Argument -- the root JSON value -- becomes the root node, which is addressed by the root selector $.

3.5.2. Dot Selector

Syntax

A dot selector starts with a dot . followed by an object's member name.

```
dot-selector    = "." dot-member-name
dot-member-name = name-first *name-char
name-first =
                      ALPHA /
                      "_"   /           ; _
                      %x80-10FFFF       ; any non-ASCII Unicode character
name-char = DIGIT / name-first

DIGIT           =  %x30-39              ; 0-9
ALPHA           =  %x41-5A / %x61-7A    ; A-Z / a-z
```

Member names containing other characters than allowed by dot-selector -- such as space ` ` and minus - characters -- MUST NOT be used with the dot-selector. (Such member names can be addressed by the index-selector instead.)

Semantics

The dot-selector selects the node of the member value corresponding to the member name from any JSON object. It selects no nodes from any other JSON value.

Note that the dot-selector 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 member names which do not encode Unicode characters.

3.5.3. Dot Wild Card Selector

Syntax

The dot wild card selector has the form .*.

```
dot-wild-selector    = "." "*"            ;  dot followed by asterisk
```

Semantics

A dot-wild-selector acts as a wild card by selecting the nodes of all member values of an object as well as all element nodes of an array. Applying the dot-wild-selector to a primitive JSON value (number, string, or true/false/null) selects no node.

3.5.4. Index Selector

Syntax

An index selector [<index>] addresses at most one object member value or at most one array element value.

```
index-selector      = "[" (quoted-member-name / element-index) "]"
```

Applying the index-selector to an object value, a quoted-member-name string is required. JSONPath allows it to be enclosed in *single* or *double* quotes.

```
quoted-member-name  = string-literal

string-literal      = %x22 *double-quoted %x22 /       ; "string"
                      %x27 *single-quoted %x27         ; 'string'

double-quoted       = unescaped /
                      %x27      /                       ; '
                      ESC %x22  /                       ; \"
                      ESC escapable

single-quoted       = unescaped /
                      %x22      /                       ; "
                      ESC %x27  /                       ; \'
                      ESC escapable

ESC                 = %x5C                              ; \  backslash

unescaped           = %x20-21 /                         ; s. RFC 8259
                      %x23-26 /                         ; omit "
                      %x28-5B /                         ; omit '
                      %x5D-10FFFF                       ; omit \

escapable           = ( %x62 / %x66 / %x6E / %x72 / %x74 / ; \b \f \n \r \t
                          ; b /         ;  BS backspace U+0008
                          ; t /         ;  HT horizontal tab U+0009
                          ; n /         ;  LF line feed U+000A
                          ; f /         ;  FF form feed U+000C
                          ; r /         ;  CR carriage return U+000D
                          "/" /          ;  /  slash (solidus)
                          "\" /          ;  \  backslash (reverse solidus)
                          (%x75 hexchar) ;  uXXXX      U+XXXX
                      )

hexchar = non-surrogate / (high-surrogate "\" %x75 low-surrogate)
non-surrogate = ((DIGIT / "A"/"B"/"C" / "E"/"F") 3HEXDIG) /
                 ("D" %x30-37 2HEXDIG )
high-surrogate = "D" ("8"/"9"/"A"/"B") 2HEXDIG
low-surrogate = "D" ("C"/"D"/"E"/"F") 2HEXDIG

HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"

; Task from 2021-06-15 interim: update ABNF later
```

Applying the index-selector to an array, a numerical element-index is required. JSONPath allows it to be negative.

```
element-index   = int                             ; decimal integer

int             = ["-"] ( "0" / (DIGIT1 *DIGIT) ) ; -  optional
DIGIT1          = %x31-39                         ; 1-9 non-zero digit
```

Notes: 1. double-quoted strings follow JSON in [RFC8259]; single-quoted strings follow an analogous pattern. 2. An element-index is an integer (in base 10, as in JSON numbers). 3. As in JSON numbers, the syntax does not allow octal-like integers with leading zeros such as 01 or -01.

Semantics

A quoted-member-name string MUST be converted to a member name by removing the surrounding quotes and replacing each escape sequence with its equivalent Unicode character, as in the table below:

Escape Sequence	Unicode Character	Description
\b	U+0008	BS backspace
\t	U+0009	HT horizontal tab
\n	U+000A	LF line feed
\f	U+000C	FF form feed
\r	U+000D	CR carriage return
\"	U+0022	quotation mark
\'	U+0027	apostrophe
\/	U+002F	slash (solidus)
\\	U+005C	backslash (reverse solidus)
\uXXXX	U+XXXX	unicode character

Table 3: Escape Sequence Replacements

The index-selector applied with a quoted-member-name to an object selects the node of the corresponding member value from it, if and only if that object has a member with that name. Nothing is selected from a value which is not a object.

Array indexing via element-index is a way of selecting a particular array element using a zero-based index. For example, selector [0] selects the first and selector [4] the fifth element of a sufficiently long array.

A negative element-index counts from the array end. For example, selector [-1] selects the last and selector [-2] selects the last but one element of an array with at least two elements.

3.5.5. Index Wild Card Selector

Syntax

The index wild card selector has the form [*].

```
index-wild-selector    = "[" "*" "]"  ;  asterisk enclosed by brackets
```

Semantics

An index-wild-selector selects the nodes of all member values of an object as well as of all elements of an array. Applying the index-wild-selector to a primitive JSON value (such as a number, string, or true/false/null) selects no node.

The index-wild-selector behaves identically to the dot-wild-selector.

3.5.6. Array Slice Selector

Syntax

The array slice selector has the form [<start>:<end>:<step>]. It selects elements starting at index <start>, ending at -- but not including -- <end>, while incrementing by step.

```
slice-selector = "[" slice-index "]"

slice-index    = ws [start] ws ":" ws [end] [ws ":" ws [step] ws]

start          = int       ; included in selection
end            = int       ; not included in selection
step           = int       ; default: 1

ws             = *( %x20 / ; Space
                    %x09 / ; Horizontal tab
                    %x0A / ; Line feed or New line
                    %x0D ) ; Carriage return
```

The slice-selector consists of three optional decimal integers separated by colons.

Semantics

The slice-selector was inspired by the slice operator of ECMAScript 4 (ES4), which was deprecated in 2014, and that of Python.

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 for this purpose is defined as:

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

The result of the array indexing expression [i] applied to an array of length len is defined to be the result of the array slicing expression [i:Normalize(i, len)+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, len)
  n_end = Normalize(end, len)

  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.

3.5.7. Descendant Selector

Syntax

The descendant selector starts with a double dot .. and can be followed by an object member name (similar to the dot-selector), by an index-selector acting on objects or arrays, or by a wild card.

```
descendant-selector = ".." ( dot-member-name      /  ; ..<name>
                             index-selector       /  ; ..[<index>]
                             index-wild-selector  /  ; ..[*]
                             "*"                     ; ..*
                           )
```

Semantics

The descendant-selector is inspired by ECMAScript for XML (E4X). It selects the node and all its descendants.

3.5.8. Union Selector

3.5.8.1. Syntax

The union selector is syntactically related to the index-selector. It contains multiple, comma separated entries.

```
union-selector = "[" ws union-entry 1*(ws "," ws union-entry) ws "]"

union-entry    =  ( quoted-member-name /
                    element-index      /
                    slice-index
                  )
```

Task (T1): This, besides slice-index, is currently one of only two places in the document that mentions whitespace. Whitespace needs to be handled throughout the ABNF syntax. Room Consensus at the 2021-06-15 interim was that JSONPath generally is generous with allowing insignificant whitespace throughout. Minimizing the impact of the many whitespace insertion points by choosing a rule name such as "S" was mentioned. Some conventions will probably help with minimizing the number of places where S needs to be inserted.

3.5.8.2. 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. Note that any node selected in more than one of the union selectors is kept as many times in the node list.

3.5.9. Filter Selector

3.5.9.1. Syntax

The filter selector has the form [?<expr>]. It works via iterating over structured values, i.e. arrays and objects.

```
filter-selector    = "[?" boolean-expr "]"
```

During iteration process each array element or object member is visited and its value -- accessible via symbol @ -- or one of its descendants -- uniquely defined by a relative path -- is tested against a boolean expression boolean-expr.

The current item is selected if and only if the result is true.

```
boolean-expr     = logical-or-expr
logical-or-expr  = logical-and-expr *("||" logical-and-expr)
                                                      ; disjunction
                                                      ; binds less tightly than conjunction
logical-and-expr = basic-expr *("&&" basic-expr)      ; conjunction
                                                      ; binds more tightly than disjunction

basic-expr   = exist-expr / paren-expr / (neg-op paren-expr) / relation-expr
exist-expr   = [neg-op] path                          ; path existence or non-existence
path         = rel-path / json-path
rel-path     = "@" *(dot-selector / index-selector)
paren-expr   = "(" boolean-expr ")"                   ; parenthesized expression
neg-op       = "!"                                    ; not operator

relation-expr = comp-expr /                           ; comparison test
                regex-expr /                          ; regular expression test
                contain-expr                          ; containment test

comp-expr    = comparable comp-op comparable
comparable   = number / string-literal /              ; primitive ...
               true / false / null /                  ; values only
               path                                   ; path value
comp-op      = "==" / "!=" /                          ; comparison ...
               "<"  / ">"  /                          ; operators
               "<=" / ">="

regex-expr   = regex-op regex
regex-op     = "=~"                                   ; regular expression match
regex        = <TO BE DEFINED>

contain-expr = containable in-op container
containable  = rel-path / json-path /                 ; path to primitive value
               number / string-literal
in-op        = " in "                                 ; in operator
container    = rel-path / json-path / array-literal   ; resolves to array
```

Notes:

*Parentheses can be used with boolean-expr for grouping. So filter selection syntax in the original proposal [?(<expr>)] is naturally contained in the current lean syntax [?<expr>] as a special case.

*Comparisons are restricted to primitive values (such as number, string, true, false, null). Comparisons with complex values will fail, i.e. no selection occurs.

* Types are not implicitly converted in comparisons. So "13 == '13'" selects no node.

* A member or element value by itself is *falsy* only, if it does not exist. Otherwise it is *truthy*, resulting in its value. To be more specific explicit comparisons are necessary. This existence test -- as an exception of the general rule -- also works with structured values.

* Regular expression tests can be applied to string values only.

* The value of the first operand (containable) of a contain-expr is compared to every single element of the RHS container. In case of a match a selection occurs. Containment tests -- like comparisons -- are restricted to primitive values. So even if a structured containable value is equal to a certain structured value in container, no selection is done.

* The value of the second operand (container) of a contain-expr needs to be resolved to an array. Otherwise nothing is selected.

The following table lists filter expression operators in order of precedence from highest (binds most tightly) to lowest (binds least tightly).

Precedence	Operator type	Syntax
5	Grouping	(...)
4	Logical NOT	!
3	Relations	== != < <= > >= =~ in
2	Logical AND	&&
1	Logical OR	||

Table 5: Filter expression operator precedence

3.5.9.2. Semantics

The filter-selector works with arrays and objects exclusively. Its result might be a list of *zero*, *one*, *multiple* or *all* of their element or member values then. Applied to other value types, it will select nothing.

Negation operator neg-op allows to test *falsiness* of values.

Type	Negation	Result	Comment
Number	!0	true	false for non-zero number
String	!"" !''	true	false for non-empty string
null	!null	true	--
true	!true	false	--
false	!false	true	--
Object	!{} !{a:0}	false	always false
Array	![] ![0]	false	always false

Table 6: Test falsiness of JSON values

Applying negation operator twice !! gives us *truthiness* of values.

Some examples:

JSON	Query	Result	Comment
{"a":1,"b":2} [2,3,4]	$[?@]	[1,2] [2,3,4]	Same as $.* or $[*]
./.	$[?@==2]	[2] [2]	Select by value.
{"a":{"b": {"c":{}}}	$[?@.b] $[?@.b.c]	[{"b": {"c":{}}]	Existence
{"key":false}	$[?index(@)=='key'] $[?index(@)==0]	[false] []	Select object member
[3,4,5]	$[?index(@)==2] $[?index(@)==17]	[5] []	Select array element
{"col":"red"}	$[?@ in ['red','green','blue']]	["red"]	Containment
{"a":{"b": {5},c:0}}	$[?@.b==5 && !@.c]	[{"b": {5},c:0}]	Existence

Table 7

4. Expression Language

Task (T2): Separate out expression language. For now, this section is a repository for ABNF taken from [RFC8259]. This needs to be deduplicated with definitions above.

```
number = [ minus ] jsint [ 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
jsint = zero / ( digit1-9 *DIGIT )
minus = %x2D               ; -
plus = %x2B                ; +
zero = %x30                ; 0

false = %x66.61.6c.73.65   ; false
null  = %x6e.75.6c.6c      ; null
true  = %x74.72.75.65      ; true
```

5. IANA Considerations

TBD: Define a media type for JSONPath expressions.

6. Security Considerations

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

7. References

7.1. 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>.

7.2. 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

Marko Mikulicic
InfluxData, Inc.
Pisa
Italy

Email: mmikulicic@gmail.com

Edward Surov
TheSoul Publishing Ltd.
Limassol
Cyprus

Email: esurov.tsp@gmail.com

Authors' Addresses

Stefan Gössner (editor)
Fachhochschule Dortmund
Sonnenstraße 96
D-44139 Dortmund
Germany

Email: stefan.goessner@fh-dortmund.de

Glyn Normington (editor)
Winchester
United Kingdom

Email: glyn.normington@gmail.com

Carsten Bormann (editor)
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

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