Workgroup: JSONPath WG
Internet-Draft: draft-ietf-jsonpath-base-06
Published: 16 August 2022
Intended Status: Standards Track
Expires: 17 February 2023
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 selecting and extracting values within a JSON (RFC 8259) value.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-jsonpath-base/.

Discussion of this document takes place on the JSON Path Working Group mailing list (mailto:jsonpath@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/jsonpath/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-jsonpath/draft-ietf-jsonpath-base.

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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 February 2023.

Copyright Notice

Table of Contents

1. Introduction

JSON [RFC8259] is a popular representation format for structured data values. JSONPath defines a string syntax for identifying values within a JSON value.

JSONPath is not intended as a replacement for, but as a more powerful companion to, JSON Pointer [RFC6901]. See Appendix B.

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 structured values, namely JSON objects and arrays, and primitive data, namely numbers and text strings as well as the special values null, true, and false.

Type: As per [RFC8259], one of the six JSON types (strings, numbers, booleans, null, objects, arrays).

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. (Not to be confused with XML element.)

Index: A non-negative integer that identifies a specific element in an array. Note that the term *indexing* is also used for accessing elements using negative integers (Section "Semantics"), and for accessing member values in an object using their member name.

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.

Children (of a node): If the node is an array, each of its elements, or if the node is an object, each of its member values (but not its member names). If the node is neither an array nor an object, it has no descendants.

Descendants (of a node): The children of the node, together with the children of its children, and so forth recursively. More formally, the descendants relation between nodes is the transitive closure of the children relation.

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].

Unicode Scalar Value: Any Unicode [UNICODE] code point except high-surrogate and low-surrogate code points. In other words, base 16 integers in either of the inclusive ranges 0 to D7FF and E000 to 10FFFF. JSON values of type string are sequences of Unicode scalar values.

Singular Path: A JSONPath expression built from selectors which each select at most one node.

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. History

This document picks up Stefan Goessner's popular JSONPath proposal dated 2007-02-21 [JSONPath-orig] and provides a normative definition for it.

Appendix A describes how JSONPath was inspired by XML's XPath [XPath].

JSONPath was intended as a light-weight companion to JSON implementations on platforms such as PHP and JavaScript, so instead of defining its own expression language like XPath did, JSONPath delegated this to the expression language of the platform. While the languages in which JSONPath is used do have significant commonalities, over time this caused non-portability of JSONPath expressions between the ensuing platform-specific dialects.

The present specification intends to remove platform dependencies and serve as a common JSONPath specification that can be used across platforms. Obviously, this means that backwards compatibility could not always be achieved; a design principle of this specification is to go with a "consensus" between implementations even if it is rough, as long as that does not jeopardize the objective of obtaining a usable, stable JSON query language.

1.3. Overview of JSONPath Expressions

JSONPath expressions are applied to a JSON value, the *argument*. Within the JSONPath expression, the abstract name $ is used to refer to the *root node* of the argument, i.e., to the argument as a whole.

JSONPath expressions can use the *dot notation*

```
$.store.book[0].title
```

or the more general *bracket notation*

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

to build paths that are input to a JSONPath implementation.

JSONPath allows the wildcard symbol * to select any member of an object or any element of an array (Section 3.4.3). The descendant operators (which start with ..) select some or all of the descendants (Section 3.4.9) of a node. The array slice syntax [start:end:step] allows selecting a regular selection of an element from an array, giving a start position, an end position, and possibly a step value that moves the position from the start to the end (Section 3.4.6).

Filter expressions are supported via the syntax ?(<boolean expr>) as in

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

Table 1 provides a quick overview of the JSONPath syntax elements.

JSONPath	Description
$	the root node (Section 3.4.1)
@	the current node: within filter selectors (Section 3.4.7)
.name	child selectors for JSON objects: dot selector (Section 3.4.2)
['name']	child selectors for JSON objects: index selector (Section 3.4.4)
..name ..[3]	descendants: descendant selector (Section 3.4.9)
*	all child member values and array elements: dot wildcard selector (Section 3.4.3), index wildcard selector (Section 3.4.5)
[3]	index (subscript) selector (Section 3.4.4): index current node as an array (from 0)
[..,..]	list selector (Section 3.4.8): allow combining selector styles
[0:100:5]	array slice selector (Section 3.4.6): start:end:step
?...	filter selector (Section 3.4.7)
()	expression: within filter selectors (Section 3.4.7)

Table 1: Overview of JSONPath

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, 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 an object, and to perform numeric comparisons of member values with a constant.

JSONPath	**Result**
$.store.book[*].author	the authors of all books in the store
$..author	all authors
$.store.*	all things in store, which are some books and a red bicycle
$.store..price	the prices of everything in the store
$..book[2]	the third book
$..book[-1]	the last book in order
	the first two books

JSONPath	Result
$..book[0,1] $..book[:2]	
$..book[?(@.isbn)]	filter all books with isbn number
$..book[?(@.price<10)]	filter all books cheaper than 10
$..*	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 JSON value.

A query **MUST** be encoded using UTF-8. The grammar for queries given in this document assumes that its UTF-8 form is first decoded into Unicode code points as described in [RFC3629]; implementation approaches that lead to an equivalent result are possible.

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 ABNF syntax in this document. A well-formed JSONPath query is valid if it also fulfills all semantic requirements posed by this document.

To be valid, integer numbers in the JSONPath query that are relevant to the JSONPath processing (e.g., index values and steps) **MUST** be within the range of exact values defined in I-JSON [RFC7493], namely within the interval $[-(2^{53})+1, (2^{53})-1]$.

To be valid, strings on the right hand side of the =~ regex matching operator need to conform to [I-D.draft-bormann-jsonpath-iregexp].

The well-formedness and the validity of JSONPath queries are independent of the JSON 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. However, the implementation **MUST NOT** silently malfunction. Specifically, if a valid JSONPath query is evaluated against a structured value whose size doesn't fit in the range of exact values, interfering with the correct interpretation of the query, the implementation **MUST** provide an indication of overflow.

(Readers familiar with the HTTP error model may be reminded of 400 type errors when pondering well-formedness and validity, while resource depletion and related errors are comparable to 500 type errors.)

The JSON value the JSONPath query is applied to is, by definition, a valid JSON value. The parsing of a JSON text into a JSON value and what happens if a JSON text does not represent valid JSON are not defined by this specification.

3.2. 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 *(S (dot-selector        /
                               dot-wild-selector   /
                               index-selector      /
                               index-wild-selector /
                               slice-selector      /
                               filter-selector     /
                               list-selector       /
                               descendant-selector))
```

The syntax and semantics of each selector is defined below.

3.3. Semantics

In this specification, the semantics of a JSONPath query define the required results and do not prescribe the internal workings of an implementation.

The semantics are that 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 presented to 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 Normalized 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 to produce the result of the selector. A node may be selected more than once and appear that number of times in the nodelist. Duplicate nodes are not removed.

The processing within a selector may execute nested queries, which conform to the semantics 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.

A syntactically valid selector **MUST NOT** produce errors. This means that some operations that might be considered erroneous, such as indexing beyond the end of an array, simply result in fewer nodes being selected.

Consider this example. 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).

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 an input node of type array all its elements (if the input note were of type object, it would select all its member values, but not the member names). 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.

3.4. Selectors

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

- Root selector $ (used at the start of a query and in expressions)
- Dot selector .<name>, used with object member names exclusively
- Dot wildcard selector .*
- Index selector [<index>], where <index> is either a (possibly negative, see Section "Semantics") array index or an object member name
- Index wildcard selector [*]
- Array slice selector [<start>:<end>:<step>], where the optional values <start>, <end>, and <step> are integer literals
- List selector [<sel1>,<sel2>,...,<selN>], holding a comma separated list of index and slice selectors
- Filter selector [?(<expr>)]
- Current item selector @ (used in expressions)
- Descendants selectors starting with a double dot ..

Note that processing the dot selector, string-valued index selector, and filter selector all potentially require matching strings against strings, with those strings coming from the JSONPath and from member names and string values in the JSON to which it is being applied. Two strings **MUST** be considered equal if and only if they are identical sequences of Unicode scalar values. In other words, normalization operations **MUST NOT** be applied to either the string from the JSONPath or from the JSON prior to comparison.

3.4.1. Root Selector

Syntax

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

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

Semantics

The root selector $ selects the root node of the argument and produces a nodelist consisting of that root node.

Examples

JSON:

```
{"k": "v"}
```

Queries:

Query	Result	Result Path	Comment
$	{"k": "v"}	$	Root node

Table 3: Root selector examples

3.4.2. 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 characters other than allowed by dot-selector -- such as space ` , minus -, or dot . 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 in its input nodelist. It selects no nodes from any other JSON value.

Examples

JSON:

```
{"j": {"k": 3}}
```

Queries:

Query	Result	Result Paths	Comment
$.j	{"k": 3}	$['j']	Named value of an object
$.j.k	3	$['j']['k']	Named value in nested object

Table 4: Dot selector examples

3.4.3. Dot Wildcard Selector

Syntax

The dot wildcard selector has the form .* as defined in the following syntax:

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

Semantics

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

Examples

JSON:

```
{
  "o": {"j": 1, "k": 2},
  "a": [5, 3]
}
```

Queries:

Query	Result	Result Paths	Comment
$.o.*	1 2	$['o']['j'] $['o']['k']	Object values
$.o.*			Alternative result

Query	Result	Result Paths	Comment
	2 1	$['o']['k'] $['o']['j']	
$.a.*	5 3	$['a'][0] $['a'][1]	Array members

Table 5: Dot wildcard selector examples

3.4.4. Index Selector

Syntax

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

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

Applying the index-selector to an object value in its input nodelist, a quoted-member-name string is required to select the corresponding member value. In contrast to JSON, the JSONPath syntax allows strings 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) U+002F
                          "\" /          ;  \  backslash (reverse solidus) U+005C
                          (%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 to select the corresponding element. JSONPath allows it to be negative (see Section "Semantics").

```
element-index   = int                              ; decimal integer

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

Notes: 1. double-quoted strings follow the JSON string syntax (Section 7 of [RFC8259]); single-quoted strings follow an analogous pattern (Section "Syntax"). 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 6: 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 that is not a object.

The index-selector applied with an element-index to an array selects an 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. Nothing is selected, and it is not an error, if the index lies outside the range of the array. Nothing is selected from a value that is not an array.

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

non-negative indexes, it is not an error if such an element does not exist; this simply means that no element is selected.

Examples

JSON:

```
{
  "o": {"j j": {"k.k": 3}},
  "a": ["a","b"],
  "'": {"@": 2}
}
```

Queries:

Query	Result	Result Paths	Comment
$.o['j j']['k.k']	3	$['o']['j j']['k.k']	Named value in nested object
$.o["j j"]["k.k"]	3	$['o']['j j']['k.k']	Named value in nested object
$.a[1]	"b"	$['a'][1]	Member of array
$.a[-2]	"a"	$['a'][0]	Member of array, from the end
$["'"]["@"]	2	$['\'']['@']	Unusual member names

Table 7: Index selector examples

3.4.5. Index Wildcard Selector

Syntax

The index wildcard selector has the form [*].

```
index-wild-selector    = "[" wildcard "]"  ;  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 (that is, a number, a string, true, false, or null) selects no node.

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

Examples

JSON:

```
{
  "o": {"j": 1, "k": 2},
  "a": [5, 3]
}
```

Queries:

Query	Result	Result Paths	Comment
$.o[*]	1 2	$['o']['j'] $['o']['k']	Object values
$.o[*]	2 1	$['o']['k'] $['o']['j']	Alternative result
$.a[*]	5 3	$['a'][0] $['a'][1]	Array members

Table 8: Index wildcard selector examples

3.4.6. 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 = "[" S slice-index S "]"

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

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

B              =    %x20 / ; Space
                    %x09 / ; Horizontal tab
                    %x0A / ; Line feed or New line
                    %x0D   ; Carriage return
S              = *B        ; optional blank space
RS             = 1*B       ; required blank space
```

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 behavior 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 that differs from the behavior of Python, which raises an error in this case.)

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

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 that 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 9: 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.

To be valid, the slice expression parameters **MUST** be in the I-JSON range of exact values, see Section 3.1.

Examples

JSON:

```
["a", "b", "c", "d", "e", "f", "g"]
```

Queries:

Query	Result	Result Paths	Comment
$[1:3]	"b" "c"	$[1] $[2]	Slice with default step
$[1:5:2]	"b" "d"	$[1] $[3]	Slice with step 2
$[5:1:-2]	"f" "d"	$[5] $[3]	Slice with negative step
$[::-1]	"g" "f" "e" "d" "c" "b" "a"	$[6] $[5] $[4] $[3] $[2] $[1] $[0]	Slice in reverse order

Table 10: Array slice selector examples

3.4.7. Filter Selector

Syntax

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

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

During the 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 boolean expression yields true.

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

basic-expr        = exist-expr /
                    paren-expr /
                    relation-expr
exist-expr        = [logical-not-op S] singular-path  ; path existence or non-existence
```

Paths in filter expressions are Singular Paths, each of which selects at most one node.

```
singular-path     = rel-singular-path / abs-singular-path
rel-singular-path = "@" *(S (dot-selector / index-selector))
abs-singular-path = root-selector *(S (dot-selector / index-selector))
```

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.

```
paren-expr        = [logical-not-op S] "(" S boolean-expr S ")"
                                                    ; parenthesized expression
logical-not-op    = "!"                             ; logical NOT operator

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

Comparisons are restricted to Singular Path values and primitive values (that is, numbers, strings, true, false, and null).

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

Alphabetic characters in ABNF are case-insensitive, so "e" can be either "e" or "E".

true, false, and null are lower-case only (case-sensitive).

```
number       = int [ frac ] [ exp ]                 ; decimal number
frac         = "." 1*DIGIT                          ; decimal fraction
exp          = "e" [ "-" / "+" ] 1*DIGIT            ; decimal exponent
true         = %x74.72.75.65                        ; true
false        = %x66.61.6c.73.65                     ; false
null         = %x6e.75.6c.6c                        ; null
```

The syntax of regular expressions in the string-literals on the right-hand side of =~ is as defined in [I-D.draft-bormann-jsonpath-iregexp].

```
regex-expr   = (singular-path / string-literal) S regex-op S regex
regex-op     = "=~"                                 ; regular expression match
regex        = string-literal                       ; I-Regexp
```

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	== != < <= > >= =~
2	Logical AND	&&
1	Logical OR	||

Table 11: Filter expression operator precedence

Semantics

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

A relative path, beginning with @, refers to the current array element or member value as the filter selector iterates over the array or object.

Existence Tests

A singular path by itself in a Boolean context is an existence test which yields true if the path selects a node and yields false if the path does not select a node. This existence test -- as an exception to the general rule -- also works with nodes with structured values.

To test the value of a node selected by a path, an explicit comparison is necessary. For example, to test whether the node selected by the path @.foo has the value null, use @.foo == null (see Section 3.5) rather than the negated existence test !@.foo (which yields false if @.foo selects a node, regardless of the node's value).

Comparisons

When a path resulting in an empty nodelist appears on either side of a comparison, the comparison yields true if and only if:

* the comparison operator is ==, >= or <= and the other side of the comparison is also a path resulting in an empty nodelist, or

* the comparison operator is != and the other side of the comparison is not also a path resulting in an empty nodelist.

When any path on either side of a comparison results in a nodelist consisting of a single node, each such path is replaced by the value of its node and then:

- a comparison using the operator == yields true if and only if the comparison is between:
  - values of the same primitive type (numbers, strings, booleans, and null) which are equal,
  - equal arrays, that is arrays of the same length where each element of the first array yields true when compared using == to the corresponding element of the second array, or
  - equal objects, that is objects where:
    - for each member of the first object with name n and value v, there is a member of the second object with name n and value w where v and w yield true when comparsed using ==, and
    - for each member of the second object with name n and value v, there is a member of the first object with name n and value w where v and w yield true when comparsed using ==.
- a comparison using the operator != yields true if and only if the comparison is not between equal values of the same type.
- a comparison using one of the operators <, <=, >, or >= yields true if and only if the comparison is between values of the same type which are both numbers or both strings and which satisfy the comparison:
  - numbers in the I-JSON [RFC7493] range of exact values **MUST** compare using the normal mathematical ordering; one or both numbers outside that range **MAY** compare using an implementation specific ordering
  - the empty string compares less than any non-empty string
  - a non-empty string compares less than another non-empty string if and only if the first string starts with a lower Unicode scalar value than the second string or if both strings start with the same Unicode scalar value and the remainder of the first string compares less than the remainder of the second string.

Note that comparisons using any of the operators <, <=, >, or >= yield false if either value being compared is an object, array, boolean, or null.

Examples

JSON:

```
{
  "obj": {"x": "y"},
  "arr": [2, 3]
}
```

Comparison	Result	Comment
$.nosuch1 == $.nosuch2	true	Empty nodelists
$.nosuch1 == 'g'	false	Empty nodelist
$.nosuch1 != $.nosuch2	false	Empty nodelists
$.nosuch1 != 'g'	true	Empty nodelist
1 <= 2	true	Numeric comparison
1 > 2	false	Strict, numeric comparison
13 == '13'	false	Type mismatch
'a' <= 'b'	true	String comparison
'a' > 'b'	false	Strict, string comparison
$.obj == $.arr	false	Type mismatch
$.obj != $.arr	true	Type mismatch
$.obj == $.obj	true	Object comparison
$.obj != $.obj	false	Object comparison
$.arr == $.arr	true	Array comparison
$.arr != $.arr	false	Array comparison
$.obj == 17	false	Type mismatch
$.obj != 17	true	Type mismatch
$.obj <= $.arr	false	Objects and arrays are not ordered
$.obj < $.arr	false	Objects and arrays are not ordered
$.obj <= $.obj	false	Objects are not ordered
$.arr <= $.arr	false	Arrays are not ordered
1 <= $.arr	false	Arrays are not ordered
1 >= $.arr	false	Arrays are not ordered
1 > $.arr	false	Arrays are not ordered
1 < $.arr	false	Arrays are not ordered
true <= true	false	Booleans are not ordered
true > true	false	Booleans are not ordered

Table 12: Comparison examples

Regular Expressions

A regular-expression test yields true if and only if the value on the left-hand side of =~ is a string value and it matches the regular expression on the right-hand side according to the semantics of [I-D.draft-bormann-jsonpath-iregexp].

The semantics of regular expressions are as defined in [I-D.draft-bormann-jsonpath-iregexp].

Boolean Operators

The logical AND, OR, and NOT operators have the normal semantics of Boolean algebra and consequently obey these laws (where P, Q, and R are any expressions with syntax logical-and-expr, T is any expression that yields true, such as 1 == 1, and F is any expression that yields false, such as 1 == 0):

Law	Expression	Equivalent expression
Associativity of OR	P || (Q || R)	(P || Q) || R
Associativity of AND	P && (Q && R)	(P && Q) && R
Commutativity of OR	P || Q	Q || R
Commutativity of AND	P && Q	Q && R
Distributivity of OR over AND	P || (Q && R)	(P || Q) && (P || R)
Distributivity of AND over OR	P && (Q || R)	(P && Q) || (P && R)
Identity for OR	P || F	P
Identity for AND	P && T	P
Annihilator for OR	P || T	T
Annihilator for AND	P && F	F
Idempotence of OR	P || P	P
Idempotence of AND	P && P	P
Absorption 1	P && (P || Q)	P
Absorption 2	P || (P && Q)	P
Complementation 1	P && !(P)	F
Complementation 2	P || !(P)	T
Double negation	!(!(P))	P
De Morgan 1	!(P) && !(Q)	!(P || Q)
De Morgan 2	!(P) || !(Q)	!(P && Q)

Table 13: Logical operator laws

Examples

JSON:

```
{
  "a": [3, 5, 1, 2, 4, 6, {"b": "ij"}, {"b": "ik"}],
  "o": {"p": 1, "q": 2, "r": 3, "s": 5, "t": {"u": 6}}
}
```

Queries:

Query	Result	Result Paths	Comment
$.a[?@>3.5]	5 4 6	$['a'] [1] $['a']	Array value comparison

Query	Result	Result Paths	Comment
		[4] $['a'][5]	
$.a[?@.b]	{"b": "ij"} {"b": "ik"}	$['a'] [6] $['a'][7]	Array value existence
$.a[?@<2 || @.b == "ik"]	1 {"b": "ik"}	$['a'] [2] $['a'][7]	Array value logical OR
$.a[?@.b =~ "i.*"]	{"b": "ij"} {"b": "ik"}	$['a'] [6] $['a'][7]	Array value regular expression
$.o[?@>1 && @<4]	2 3	$['o'] ['q'] $['o'] ['r']	Object value logical AND
$.o[?@>1 && @<4]	3 2	$['o'] ['r'] $['o'] ['q']	Alternative result
$.o[?@.u || @.x]	{"u": 6}	$['o'] ['t']	Object value logical OR
$.a[?(@.b == $.x)]	3 5 1 2 4 6	$['a'] [0] $['a'] [1] $['a'] [2] $['a'] [3] $['a'][4]	Comparison of paths with no values
$[?(@ == @)]			Comparison of structured values

Table 14: Filter selector examples

3.4.8. List Selector

The list selector allows combining member names, array indices, slices, and filters in a single selector.

Note: The list selector was called "union selector" in [JSONPath-orig], as it was intended to solve use cases addressed by the union selector in XPath. However, the term "union" has the connotation of a set operation that involves merging input sets while avoiding duplicates, so the concept was renamed into "list selector".

Syntax

The list selector is syntactically related to the dot-selector, index-selector, slice-selector, and the filter-selector. It contains two or more entries, separated by commas.

```
list-selector  = "[" S list-entry 1*(S "," S list-entry) S "]"

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

Semantics

A list selector selects the nodes that are selected by at least one of the selector entries in the list and yields the concatenation of the lists (in the order of the selector entries) of nodes selected by the selector entries. Note that any node selected in more than one of the selector entries is kept as many times in the nodelist.

To be valid, integer values in the element-index and slice-index components **MUST** be in the I-JSON [RFC7493] range of exact values, see Section 3.1.

Examples

JSON:

```
["a", "b", "c", "d", "e", "f", "g"]
```

Queries:

Query	Result	Result Paths	Comment
$[0, 3]	"a" "d"	$[0] $[3]	Indices
$[0:2, 5]	"a" "b" "f"	$[0] $[1] $[5]	Slice and index
$[0, 0]	"a" "a"	$[0] $[0]	Duplicated entries

Table 15: List selector examples

3.4.9. Descendant Selectors

Syntax

The descendant selectors start with a double dot .. and can be followed by an object member name (similar to the dot-selector), a wildcard (similar to the dot-wild-selector), an index-selector, index-wild-selector, filter-selector, or list-selector acting on objects or arrays, or a slice-selector acting on arrays.

```
descendant-selector = ".." ( dot-member-name      /  ; ..<name>
                             wildcard             /  ; ..*
                             index-selector       /  ; ..[<index>]
                             index-wild-selector  /  ; ..[*]
                             slice-selector       /  ; ..[<slice-index>]
                             filter-selector      /  ; ..[<filter>]
                             list-selector           ; ..[<list-entry>,...]
                           )
```

Note that .. on its own is not a valid selector.

Semantics

A descendant-selector selects certain descendants of a node:

- the ..<name> form (and the ..[<index>] form where <index> is a quoted-member-name) selects those descendants that are member values of an object with the given member name.
- the ..[<index>] form, where <index> is an element-index, selects those descendants that are array elements with the given index.
- the ..[<slice-index>] form selects those descendants that are array elements selected by the given slice.
- the ..[<filter>] form selects those descendants that are array elements or object values selected by the given filter.
- the ..[*] and ..* forms select all the descendants.

An *array-sequenced preorder* of the descendants of a node is a sequence of all the descendants in which:

- nodes of any array appear in array order,
- nodes appear immediately before all their descendants.

This definition does not stipulate the order in which the children of an object appear, since JSON objects are unordered.

The resultant nodelist of a descendant-selector applied to a node must be a sub-sequence of an array-sequenced preorder of the descendants of the node.

Examples

JSON:

```
{
  "o": {"j": 1, "k": 2},
  "a": [5, 3, [{"j": 4}]]
}
```

Queries:

Query	Result	Result Paths	Comment
$..j	1 4	$['o']['j'] $['a'][2][0]['j']	Object values
$..j	4 1	$['a'][2][0]['j'] $['o']['j']	Alternative result
$..[0]	5 {"j": 4}	$['a'][0] $['a'][2][0]	Array values
$..[0]	{"j": 4} 5	$['a'][2][0] $['a'][0]	Alternative result
$..[*]	{"j": 1, "k" : 2} [5, 3, [{"j": 4}]] 1 2 5 3 [{"j": 4}] {"j": 4} 4	$['o'] $['a'] $['o']['j'] $['o']['k'] $['a'][0] $['a'][1] $['a'][2] $['a'][2][0] $['a'][2][0]['j']	All values
$..*	[5, 3, [{"j": 4}]] {"j": 1, "k" : 2} 2 1 5 3 [{"j": 4}] {"j": 4} 4	$['a'] $['o'] $['o']['k'] $['o']['j'] $['a'][0] $['a'][1] $['a'][2] $['a'][2][0] $['a'][2][0]['j']	All values

Table 16: Descendant selector examples

Note: The ordering of the results for the $..[*] and $..* examples above is not guaranteed, except that:

* {"j": 1, "k": 2} must appear before 1 and 2,

* [5, 3, [{"j": 4}]] must appear before 5, 3, and [{"j": 4}],

* 5 must appear before 3 which must appear before [{"j": 4}],

* 5 and 3 must appear before {"j": 4} and 4,

* [{"j": 4}] must appear before {"j": 4}, and

* {"j": 4} must appear before 4.

3.5. Semantics of null

Note that JSON null is treated the same as any other JSON value: it is not taken to mean "undefined" or "missing".

Examples

JSON:

```
{"a": null, "b": [null], "c": [{}], "null": 1}
```

Queries:

Query	Result	Result Paths	Comment
$.a	null	$['a']	Object value
$.a[0]			null used as array
$.a.d			null used as object
$.b[0]	null	$['b'][0]	Array value
$.b[*]	null	$['b'][0]	Array value
$.b[?@]	null	$['b'][0]	Existence
$.b[?@==null]	null	$['b'][0]	Comparison
$.c[?(@.d==null)]			Comparison with "missing" value
$.null	1	$['null']	Not JSON null at all, just a string as object key

Table 17: Examples involving (or not involving) null

3.6. Normalized Paths

A Normalized Path is a JSONPath with restricted syntax that identifies a node by providing a query that results in exactly that node. For example, the JSONPath expression $.book[?(@.price<10)] could select two values with Normalized Paths $['book'][3] and $

['book'][5]. For a given JSON value, there is a one to one correspondence between the value's nodes and the Normalized Paths that identify these nodes.

A JSONPath implementation may output Normalized Paths instead of, or in addition to, the values identified by these paths.

Since bracket notation is more general than dot notation, it is used to construct Normalized Paths. Single quotes are used to delimit string member names. This reduces the number of characters that need escaping when Normalized Paths appear as strings (which are delimited with double quotes) in JSON texts.

The syntax of Normalized Paths is restricted so that there is one and only one way of representing any given Normalized Path. Putting this another way, for any two distinct Normalized Paths, a JSON value exists that will yield distinct results when the Normalized Paths are applied to it.

Certain characters are escaped, in one and only one way; all other characters are unescaped.

Normalized Paths are Singular Paths. Not all Singular Paths are Normalized Paths: $[-3], for example, is a Singular Path, but not a Normalized Path.

```
normalized-path           = root-selector *(normal-index-selector)
normal-index-selector     = "[" (normal-quoted-member-name / normal-element-index) "]"
normal-quoted-member-name = %x27 *normal-single-quoted %x27 ; 'string'
normal-single-quoted      = normal-unescaped /
                            ESC normal-escapable
normal-unescaped          = %x20-26 /                      ; omit control codes
                            %x28-5B /                      ; omit '
                            %x5D-10FFFF                    ; omit \
normal-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
                                "'" /         ;  ' apostrophe U+0027
                                "\" /         ;  \ backslash (reverse solidus) U+005C
                                (%x75 normal-hexchar) ;  certain values u00xx U+00XX
                            )
normal-hexchar            = "0" "0"
                            (
                              ("0" %x30-37) / ; "00"-"07"
                              ("0" %x62) /    ; "0b"      ; omit U+0008-U+000A
                              ("0" %x65-66) /  ; "0e"-"0f" ; omit U+000C-U+000D
                              ("1" normal-HEXDIG)
                            )
normal-HEXDIG             = DIGIT / %x61-66   ; "0"-"9", "a"-"f"
normal-element-index      = "0" / (DIGIT1 *DIGIT) ; non-negative decimal integer
```

Examples

Path	Normalized Path	Comment
$.a	$['a']	Object value
$[1]	$[1]	Array index
$.a.b[1:2]	$['a']['b'][1]	Nested structure
$["\u000B"]	$['\u000b']	Unicode escape
$["\u0061"]	$['a']	Unicode character
$["\u00b1"]	$['±'] (U+0024 U+005B U+0027 U+00B1 U+0027 U+005D)	Unicode character

Table 18: Normalized Path examples

$["\u00b1"] is normalized into $['±'] (noise in the table and lack of typewriter font is due to RFCXMLv3 limitations).

4. IANA Considerations

4.1. Registration of Media Type application/jsonpath

IANA is requested to register the following media type [RFC6838]:

Type name: application

Subtype name: jsonpath

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary (UTF-8)

Security considerations: See the Security Considerations section of RFCXXXX.

Interoperability considerations: N/A

Published specification: RFCXXXX

Applications that use this media type: Applications that need to convey queries in JSON data

Fragment identifier considerations: N/A

Additional information:

Deprecated alias names for this type: N/A

Magic number(s): N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person & email address to contact for further information: iesg@ietf.org

Intended usage: COMMON

Restrictions on usage: N/A

Author: JSONPath WG

Change controller: IESG

Provisional registration? (standards tree only): no

5. Security Considerations

Security considerations for JSONPath can stem from

- attack vectors on JSONPath implementations, and
- the way JSONPath is used in security-relevant mechanisms.

5.1. Attack vectors on JSONPath Implementations

Historically, JSONPath has often been implemented by feeding parts of the query to an underlying programming language engine, e.g., JavaScript. This approach is well known to lead to injection attacks and would require perfect input validation to prevent these attacks (see Section 12 of [RFC8259] for similar considerations for JSON itself). Instead, JSONPath implementations need to implement the entire syntax of the query without relying on the parsers of programming language engines.

Attacks on availability may attempt to trigger unusually expensive runtime performance exhibited by certain implementations in certain cases. (See Section 10 of [RFC8949] for issues in hash-table implementations, and Section 8 of [I-D.draft-bormann-jsonpath-iregexp] for performance issues in regular expression implementations.) Implementers need to be aware that good average performance is not sufficient as long as an attacker can choose to submit specially crafted JSONPath queries or arguments that trigger surprisingly high, possibly exponential, CPU usage or, for example via a naive recursive implementation of the descendant selector, stack overflow. Implementations need to have appropriate resource management to mitigate these attacks.

5.2. Attacks on Security Mechanisms that Employ JSONPath

Where JSONPath is used as a part of a security mechanism, attackers can attempt to provoke unexpected or unpredictable behavior, or take advantage of differences in behavior between JSONPath implementations.

Unexpected or unpredictable behavior can arise from an argument with certain constructs described as unpredictable by [RFC8259]. Predictable behavior can be expected, except in relation to the ordering of objects, for any argument conforming with [RFC7493].

Other attacks can target the behavior of underlying technologies such as UTF-8 (see Section 10 of [RFC3629]) and the Unicode character set.

6. References

6.1. Normative References

[I-D.draft-bormann-jsonpath-iregexp] Bormann, C. and T. Bray, "I-Regexp: An Interoperable Regexp Format", Work in Progress, Internet-Draft, draft-bormann-jsonpath-iregexp-04, 25 April 2022, <https://www.ietf.org/archive/id/draft-bormann-jsonpath-iregexp-04.txt>.

[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>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[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>.

[UNICODE] The Unicode Consortium, "The Unicode® Standard: Version 14.0 - Core Specification", September 2021, <https://www.unicode.org/versions/Unicode14.0.0/UnicodeStandard-14.0.pdf>.

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/>.

[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>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

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

[XPath] Berglund, A., Ed., Chamberlin, D., Ed., Simeon, J., Ed., Robie, J., Ed., Fernandez, M., Ed., Kay, M., Ed., and S. Boag, Ed., "XML Path Language (XPath) 2.0 (Second Edition)", W3C REC REC-xpath20-20101214, W3C REC-xpath20-20101214, 14 December 2010, <https://www.w3.org/TR/2010/REC-xpath20-20101214/>.

Appendix A. Inspired by XPath

This appendix is informative.

At the time JSONPath was invented, XML was noted for the availability of powerful tools to analyze, 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.

(Note that XPath has evolved since 2007, and recent versions even nominally support operating inside JSON values. This appendix only discusses the more widely used version of XPath that was available in 2007.)

JSONPath picks up the overall feeling of XPath, but maps the concepts to syntax (and partially semantics) that would be familiar to someone using JSON in a dynamic language.

E.g., in popular dynamic programming languages such as JavaScript, Python and PHP, the semantics of the XPath expression

```
/store/book[1]/title
```

can be realized in the expression

```
x.store.book[0].title
```

or, in bracket notation,

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

with the variable x holding the argument.

The JSONPath language was designed to:

- be naturally based on those language characteristics;
- cover only the most essential parts of XPath 1.0;
- be lightweight in code size and memory consumption;
- be runtime efficient.

A.1. JSONPath and XPath

JSONPath expressions apply to JSON values in the same way as XPath expressions are used in combination with an XML document. JSONPath uses $ to refer to the root node of the argument, similar to XPath's / at the front.

JSONPath expressions move further down the hierarchy using *dot notation* ($.store.book[0].title) or the *bracket notation* ($['store']['book'][0]['title']), a lightweight/limited, and a more heavyweight syntax replacing XPath's / within query expressions.

Both JSONPath and XPath use * for a wildcard. The descendant operators, starting with .., borrowed from [E4X], are similar to XPath's //. The array slicing construct [start:end:step] is unique to JSONPath, inspired by [SLICE] from ECMASCRIPT 4.

Filter expressions are supported via the syntax ?(<boolean expr>) as in

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

Table 19 extends Table 1 by providing a comparison with similar XPath concepts.

XPath	JSONPath	Description
/	$	the root XML element
.	@	the current XML element
/	. or []	child operator
..	n/a	parent operator
//	..name, .. [index], ..*, or .. [*]	descendants (JSONPath borrows this syntax from E4X)
*	*	wildcard: All XML elements regardless of their names
@	n/a	attribute access: JSON values do not have attributes
[]	[]	subscript operator used to iterate over XML element collections and for predicates
|	[,]	Union operator (results in a combination of node sets); called list operator in JSONPath, allows combining member names, array indices, and slices
n/a	[start:end:step]	array slice operator borrowed from ES4
[]	?()	applies a filter (script) expression
seamless	n/a	expression engine
()	n/a	grouping

Table 19: XPath syntax compared to JSONPath

For further illustration, Table 20 shows some XPath expressions and their JSONPath equivalents.

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[-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 20: Example XPath expressions and their JSONPath equivalents

XPath has a lot more functionality (location paths in unabbreviated syntax, operators and functions) than listed in this comparison. Moreover, there are significant differences in 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.

Appendix B. JSON Pointer

This appendix is informative.

JSONPath is not intended as a replacement for, but as a more powerful companion to, JSON Pointer [RFC6901]. The purposes of the two standards are different.

JSON Pointer is for identifying a single value within a JSON value whose structure is known.

JSONPath can identify a single value within a JSON value, for example by using a Normalized Path. But JSONPath is also a query syntax that can be used to search for and extract multiple values from JSON values whose structure is known only in a general way.

A Normalized JSONPath can be converted into a JSON Pointer by converting the syntax, without knowledge of any JSON value. The inverse is not generally true: a numeric path component in a JSON Pointer may identify a member of a JSON object or may index an

array. For conversion to a JSONPath query, knowledge of the structure of the JSON value is needed to distinguish these cases.

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