JSONPath: Query expressions for JSON

Abstract

JSONPath defines a string syntax for selecting and extracting JSON (RFC 8259) values from a JSON 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/. Subscribe at https://www.ietf.org/mailman/listinfo/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 10 August 2023.
1. Introduction
   1.1. Terminology
      1.1.1. JSON Values as Trees of Nodes
   1.2. History
   1.3. JSON Values
   1.4. Overview of JSONPath Expressions
      1.4.1. Identifiers
      1.4.2. Segments
      1.4.3. Selectors
      1.4.4. Summary
   1.5. JSONPath Examples

2. JSONPath Syntax and Semantics
   2.1. Overview
   2.2. Syntax
   2.3. Semantics
      2.3.1. Worked Example
   2.4. Root Identifier
   2.5. Selectors
      2.5.1. Name Selector
      2.5.2. Wildcard Selector
      2.5.3. Index Selector
      2.5.4. Array Slice selector
      2.5.5. Filter selector
   2.6. Function Extensions
      2.6.1. Type System for Function Expressions
      2.6.2. Type Correctness of Function Expressions
      2.6.3. length Function Extension
      2.6.4. count Function Extension
      2.6.5. match Function Extension
      2.6.6. search Function Extension
   2.7. Segments
      2.7.1. Child Segment
1. Introduction

JSON [RFC8259] is a popular representation format for structured data values. JSONPath defines a string syntax for selecting and extracting JSON values from 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 different kinds of values as in Section 1 of [RFC8259]; JSON Objects and Arrays are structured, all other values are primitive. 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 document are defined below.

**Value:** As per [RFC8259], a structure conforming 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. [RFC8259] focuses on the textual representation of JSON values and does not fully define the value abstraction assumed here.

**Member:** A name/value pair in an object. (A member is not itself a value.)

**Name:** The name (a string) in a name/value pair constituting a member. This is also used in [RFC8259], but that specification does not formally define it. It is included here for completeness.

**Element:** A value in a JSON array.

**Index:** An integer that identifies a specific element in an array.

**Query:** Short name for a JSONPath expression.

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

**Location:** the position of a value within the argument. This can be thought of as a sequence of names and indexes navigating to the value through the objects and arrays in the argument, with the empty sequence indicating the argument itself. A location can be represented as a Normalized Path (defined below).

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

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

**Root Node Identifier:** The expression $ which refers to the root node of the argument.

**Current Node Identifier:** The expression @ which refers to the current node in the context of the evaluation of a filter expression (described later).

**Children (of a node):** If the node is an array, the nodes of its elements. If the node is an object, the nodes of its member
values. If the node is neither an array nor an object, it has no children.

**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.

**Depth (of a descendant node within a value):** The number of ancestors of the node within the value. The root node of the value has depth zero, the children of the root node have depth one, their children have depth two, and so forth.

**Segment:** One of the constructs which select children ([]) or descendants (..[]) of an input value.

**Nodelist:** A list of nodes. While a nodelist can be represented in JSON, e.g. as an array, this document does not require or assume any particular representation.

**Normalized Path:** A simple form of JSONPath expression that identifies a node in a value 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, integers in either of the inclusive base 16 ranges 0 to D7FF and E000 to 10FFFF. JSON string values are sequences of Unicode scalar values.

**Singular Nodelist:** A nodelist containing at most one node.

**Singular Path:** A JSONPath expression built from segments each of which, regardless of the input value, produces a Singular Nodelist.

**Selector:** A single item within a segment that takes the input value and produces a nodelist consisting of child nodes of the input value.

### 1.1.1. JSON Values as Trees of Nodes

This document models the argument as a tree of JSON values, each with its own node. A node is either the root node or one of its descendants.

This document models the result of applying a query to the argument as a nodelist (a list of nodes).
Nodes are the selectable parts of the argument. The only parts of an object that can be selected by a query are the member values. Member names and members (name/value pairs) cannot be selected. Thus, member values have nodes, but members and member names do not. Similarly, member values are children of an object, but members and member names are not.

1.2. History

This section is informative.

This document picks up Stefan Gössner's popular JSONPath proposal dated 2007-02-21 [JSONPath-orig], builds on the experience from the widespread deployment of its implementations, and provides a normative specification 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 in programming languages such as PHP and JavaScript, so instead of defining its own expression language, like XPath did, JSONPath delegated parts of a query to the underlying runtime, e.g., JavaScript's eval() function. As JSONPath was implemented in more environments, JSONPath expressions became decreasingly portable. For example, regular expression processing was often delegated to a convenient regular expression engine.

This document aims to remove such implementation-specific dependencies and serve as a common JSONPath specification that can be used across programming languages and environments. This means that backwards compatibility is not always achieved; a design principle of this document 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. JSON Values

The JSON value a JSONPath query is applied to is, by definition, a valid JSON value. A JSON value is often constructed by parsing a JSON text.

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 document. Sections 4 and 8 of [RFC8259] identify specific situations that may conform to the grammar for JSON texts but are not interoperable uses of JSON, as they may cause unpredictable behavior. This document does not attempt to define predictable behavior for JSONPath queries in these situations.
Specifically, the "Semantics" subsections of Sections 2.5.1, 2.5.2, 2.5.5, and 2.7.2 describe behavior that becomes unpredictable when the JSON value for one of the objects under consideration was constructed out of JSON text that exhibits multiple members for a single object that share the same member name ("duplicate names", see Section 4 of [RFC8259]). Also, selecting a child by name (Section 2.5.1) and comparing strings (Section "Comparisons" in Section 2.5.5) assume these strings are sequences of Unicode scalar values, becoming unpredictable if they are not (Section 8.2 of [RFC8259]).

1.4. Overview of JSONPath Expressions

This section is informative.

A JSONPath expression is applied to a JSON value, known as the argument. The output is a nodelist.

A JSONPath expression consists of an identifier followed by a series of zero or more segments each of which contains one or more selectors.

1.4.1. Identifiers

The root node identifier $ refers to the root node of the argument, i.e., to the argument as a whole.

The current node identifier @ refers to the current node in the context of the evaluation of a filter expression (described later).

1.4.2. Segments

Segments select children ([]) or descendants (..[]) of an input value.

Segments can use bracket notation, for example:

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

or the more compact dot notation, for example:

$.store.book[0].title

A JSONPath expression may use a combination of bracket and dot notations.

This document treats the bracket notations as canonical and defines the shorthand dot notation in terms of bracket notation. Examples and descriptions use shorthands where convenient.
1.4.3. Selectors

A name selector, e.g. 'name', selects a named child of an object.

An index selector, e.g. 3, selects an indexed child of an array.

A wildcard * ([Section 2.5.2]) in the expression [*] selects all children of a node and in the expression ..[*] selects all descendants of a node.

An array slice start:end:step ([Section 2.5.4]) selects a series of elements from an array, giving a start position, an end position, and an optional step value that moves the position from the start to the end.

Filter expressions ?<boolean expr> select certain children of an object or array, as in:

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

1.4.4. Summary

Table 1 provides a brief overview of JSONPath syntax.

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>root node identifier ([Section 2.4])</td>
</tr>
<tr>
<td>@</td>
<td>current node identifier ([Section 2.5.5]) (valid only within filter selectors)</td>
</tr>
<tr>
<td>[&lt;selectors&gt;]</td>
<td>child segment ([Section 2.7.1]) selects zero or more children of a node; contains one or more selectors, separated by commas</td>
</tr>
<tr>
<td>..</td>
<td>descendant segment ([Section 2.7.2]): selects zero or more descendants of a node; contains one or more selectors, separated by commas</td>
</tr>
<tr>
<td>..name</td>
<td>shorthand for ..['name']</td>
</tr>
<tr>
<td>..*</td>
<td>shorthand for ..[*]</td>
</tr>
<tr>
<td>..[&lt;selectors&gt;]</td>
<td>name selector ([Section 2.5.1]): selects a named child of an object</td>
</tr>
<tr>
<td>'name'</td>
<td>wildcard selector ([Section 2.5.1]): selects all children of a node</td>
</tr>
<tr>
<td>3</td>
<td>index selector ([Section 2.5.3]): selects an indexed child of an array (from 0)</td>
</tr>
<tr>
<td>0:100:5</td>
<td>array slice selector ([Section 2.5.4]): start:end:step for arrays</td>
</tr>
<tr>
<td>?&lt;expr&gt;</td>
<td>filter selector ([Section 2.5.5]): selects particular children using a boolean expression</td>
</tr>
</tbody>
</table>
Syntax | Description
--- | ---
`length(@.foo)` | **function extension** ([Section 2.6](#)): invokes a function in a filter expression

Table 1: Overview of JSONPath syntax

1.5. JSONPath Examples

This section is informative. It provides examples of JSONPath expressions.

The examples are based on the simple JSON value shown in Figure 1, representing a bookstore (that also has a bicycle).

```json
{ "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": 399
  }
}
```

Figure 1: Example JSON value
Table 2 shows some JSONPath queries that might be applied to this example and their intended results.

<table>
<thead>
<tr>
<th>JSONPath</th>
<th>Intended result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.store.book[*].author</td>
<td>the authors of all books in the store</td>
</tr>
<tr>
<td>$.author</td>
<td>all authors</td>
</tr>
<tr>
<td>$.store.*</td>
<td>all things in store, which are some books and a red bicycle</td>
</tr>
<tr>
<td>$.store..price</td>
<td>the prices of everything in the store</td>
</tr>
<tr>
<td>$.book[-1]</td>
<td>the last book in order</td>
</tr>
<tr>
<td>$.book[0,1]</td>
<td>the first two books</td>
</tr>
<tr>
<td>$.book[:2]</td>
<td></td>
</tr>
<tr>
<td>$.book[?(@.isbn)]</td>
<td>all books with an ISBN number</td>
</tr>
<tr>
<td>$.book[?(@.price&lt;10)]</td>
<td>all books cheaper than 10</td>
</tr>
<tr>
<td>$.*</td>
<td>all member values and array elements</td>
</tr>
</tbody>
</table>

Table 2: Example JSONPath expressions and their intended results when applied to the example JSON value

2. JSONPath Syntax and Semantics

2.1. Overview

A JSONPath expression is a string which, when applied to a JSON value, the argument, selects zero or more nodes of the argument and outputs these nodes as a nodelist.

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, which are:

1. 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 \([-2^{53}+1, 2^{53}-1]\).

2. Uses of function extensions must be correctly typed, as described in Section 2.6.
A JSONPath implementation **MUST** raise an error for any query which is not well-formed and valid. The well-formedness and the validity of JSONPath queries are independent of the JSON value the query is applied to; no further errors relating to the well-formedness and the validity of a JSONPath query 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 this document. However, the implementation **MUST NOT** silently malfunction. Specifically, if a valid JSONPath query is evaluated against a structured value whose size does not 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.)

### 2.2. Syntax

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

```
json-path           = root-identifier segments
segments            = *(S segment)
```

The syntax and semantics of segments are defined in [Section 2.7](#).

### 2.3. Semantics

In this document, the semantics of a JSONPath query define the required results and do not prescribe the internal workings of an implementation. This document may describe semantics in a procedural step-by-step fashion, but such descriptions are normative only in the sense that any implementation **MUST** produce an identical result, but not in the sense that implementors are required to use the same algorithms.

The semantics are that a valid query is executed against a value, the *argument*, and produces a nodelist (i.e., a list of zero or more nodes of the value).

The query is a root identifier followed by a sequence of zero or more segments, each of which is applied to the result of the previous root identifier or segment and provides input to the next segment. These results and inputs take the form of a nodelist.
Segments can be added to a query to drill further into the structure of the input value.

The nodelist resulting from the root identifier contains a single node, the argument. The nodelist resulting from the last segment 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 an empty nodelist is a valid query result.

A segment operates on each of the nodes in its input nodelist in turn, and the resultant nodelists are concatenated to produce the result of the segment. A node may be selected more than once and appears that number of times in the nodelist. Duplicate nodes are not removed.

A syntactically valid segment MUST NOT produce errors when executing the query. This means that some operations that might be considered erroneous, such as using an index lying outside the range of an array, simply result in fewer nodes being selected.

As a consequence of this approach, if any of the segments produces an empty nodelist, then the whole query produces an empty nodelist.

If a query may produce a nodelist with more than one possible ordering, a particular implementation may also produce distinct orderings in successive runs of the query.

2.3.1. Worked Example

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 segments: .a, ["b"], and .b.

Firstly, $ produces a nodelist consisting of just the argument.

Next, .a selects from any object input node 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 array input node all its elements (for an object input node, 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 object input node 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.

2.4. Root Identifier

Syntax

Every JSONPath query (except those inside filter expressions, see
Section 2.5.5) MUST begin with the root identifier $.

root-identifier  = "$"

Semantics

The root identifier $ represents the root node of the argument and
produces a nodelist consisting of that root node.

Examples

JSON:

{"k": "v"}

Queries:

<table>
<thead>
<tr>
<th>Query</th>
<th>Result Path</th>
<th>Result Path</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>{&quot;k&quot;: &quot;v&quot;}</td>
<td>$</td>
<td>Root node</td>
</tr>
</tbody>
</table>

Table 3: Root identifier examples

2.5. Selectors

Selectors appear only inside child segments (Section 2.7.1) and
descendant segments (Section 2.7.2).

A selector produces a nodelist consisting of zero or more children
of the input value.

There are various kinds of selectors which produce children of
objects, children of arrays, or children of either objects or
arrays.
The syntax and semantics of each kind of selector are defined below.

### 2.5.1. Name Selector

**Syntax**

A name selector '<name>' selects at most one object member value.

In contrast to JSON, the JSONPath syntax allows strings to be enclosed in *single or double quotes.*
name-selector = 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 / ; see RFC 8259
            %x23-26 /
            %x28-5B /
            %x5D-10FFFF

escapable = %x62 / ; b BS backspace U+0008
            %x66 / ; f FF form feed U+000C
            %x6E / ; n LF line feed U+000A
            %x72 / ; r CR carriage return U+000D
            %x74 / ; t HT horizontal tab U+0009
            "/" / ; / slash (solidus) U+002F
            "\" / ; \ backslash (reverse solidus) U+005C
            (%x75 hexchar) ; uXXXX U+XXXX

oxchar = 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"

Note: double-quoted strings follow the JSON string syntax ([Section 7 of [RFC8259]]); single-quoted strings follow an analogous pattern ([Section "Syntax"]). No attempt was made to improve on this syntax, so characters with scalar values above 0x10000, such as U+1F914 (" ", THINKING FACE), need to be represented by a pair of surrogate escapes ("\uD83E\uDD14" in this case).
Semantics

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

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

Table 4: Escape Sequence Replacements

Applying the name-selector to an object node selects a member value whose name equals the member name M, or selects nothing if there is no such member value. Nothing is selected from a value that is not an object.

Note that processing the name selector requires comparing the member name string M with member name strings in the JSON to which the selector 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 member name string M from the JSONPath or to the member name strings in the JSON prior to comparison.

Examples

JSON:

```
{
  "o": {"j j": {"k.k": 3}},
  "": {"": 2}
}
```

Queries:

The following examples show the name selector in use by child segments.
2.5.2. Wildcard Selector

Syntax

The wildcard selector consists of an asterisk.

```
wildcard-selector  = "*"
```

Semantics

A wildcard selector selects the nodes of all children of an object or array. The order in which the children of an object appear in the resultant nodelist is not stipulated, since JSON objects are unordered. Children of an array appear in array order in the resultant nodelist.

The wildcard selector selects nothing from a primitive JSON value (that is, a number, a string, true, false, or null).

Examples

JSON:

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

Queries:

The following examples show the wildcard selector in use by a child segment.

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Result Paths</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$[*]</code></td>
<td><code>{&quot;j&quot;: 1, &quot;k&quot;: 2}</code></td>
<td><code>$[o]</code> {&quot;j&quot;: 1, &quot;k&quot;: 2}</td>
<td>Object values</td>
</tr>
<tr>
<td><code>.o[*]</code></td>
<td><code>[5, 3]</code></td>
<td><code>['a']</code></td>
<td>Object values</td>
</tr>
<tr>
<td>Query</td>
<td>Result</td>
<td>Paths</td>
<td>Comment</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>$.o[*]</td>
<td>2</td>
<td>[&quot;o&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>[&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;o&quot;][&quot;k&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;o&quot;]</td>
<td></td>
</tr>
<tr>
<td>$.o[*]</td>
<td>1</td>
<td>[&quot;k&quot;]</td>
<td>Alternative result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;o&quot;][&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;o&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td>$.o[*]</td>
<td>2</td>
<td>[&quot;o&quot;]</td>
<td>Non-deterministic ordering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;k&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;o&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[&quot;k&quot;]</td>
<td></td>
</tr>
<tr>
<td>$.o[*]</td>
<td>1</td>
<td>[&quot;o&quot;][&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td>$.a[*]</td>
<td>5</td>
<td>[&quot;a&quot;]</td>
<td>Array members</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>[&quot;a&quot;][0]</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Wildcard selector examples

The example above with the query $.o[*] shows that the wildcard selector may produce nodelists in distinct orders each time it appears in the child segment, when it is applied to an object node with two or more members (but not when it is applied to object nodes with fewer than two members or to array nodes).

### 2.5.3. Index Selector

**Syntax**

An index selector `<index>` matches at most one array element value.

```
index-selector = int ; decimal integer

int = "0" / (["-" DIGIT1 *DIGIT) ; - optional

DIGIT1 = %x31-39 ; 1-9 non-zero digit
```

Applying the numerical index-selector selects the corresponding element. JSONPath allows it to be negative (see Section "Semantics").

To be valid, the index selector value **MUST** be in the I-JSON range of exact values, see Section 2.1.

**Notes:**
1. An index-selector is an integer (in base 10, as in JSON numbers).
2. As in JSON numbers, the syntax does not allow octal-like integers with leading zeros such as 01 or -01.
Semantics

A non-negative index-selector applied to an array selects an array element using a zero-based index. For example, the selector 0 selects the first and the selector 4 selects 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 index-selector counts from the array end. For example, the selector -1 selects the last and the 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:

["a","b"]

Queries:

The following examples show the index selector in use by a child segment.

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Result Paths</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[-2]</td>
<td>&quot;a&quot;</td>
<td>$[0]</td>
<td>Element of array, from the end</td>
</tr>
</tbody>
</table>

Table 7: Index selector examples

2.5.4. Array Slice selector

Syntax

The array slice selector has the form <start>:<end>:<step>. It matches elements from arrays starting at index <start>, ending at — but not including — <end>, while incrementing by step with a default of 1.
The slice selector consists of three optional decimal integers separated by colons.

To be valid, the integers provided MUST be in the I-JSON range of exact values, see Section 2.1.

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 informative.

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.
Normative Semantics

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. In the rest of this section, len denotes 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:

<table>
<thead>
<tr>
<th>Condition</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>step &gt;= 0</td>
<td>0</td>
<td>len</td>
</tr>
<tr>
<td>step &lt; 0</td>
<td>len - 1</td>
<td>len - 1</td>
</tr>
</tbody>
</table>

Table 8: 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 index expression i applied to an array of length len is the result of the array slicing expression Normalize(i, len):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:
The slice expression selects elements with indices between the lower and upper bounds. In the following pseudocode, a(i) is the i+1th element of the array a (i.e., a(0) is the first element, a(1) the second, and so forth).

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

The slice expression selects elements with indices between the lower and upper bounds. In the following pseudocode, a(i) is the i+1th element of the array a (i.e., a(0) is the first element, a(1) the second, and so forth).

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)

    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.

Examples

    JSON:

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

    Queries:

    The following examples show the array slice selector in use by a child segment.
Table 9: Array slice selector examples

2.5.5. Filter selector

Syntax

The filter selector has the form ?<expr>. It iterates over structured values, i.e., arrays and objects.

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

During the iteration process the node of each array element or object member value being visited is known as the current node. A boolean expression, usually involving the current node, is evaluated and the current node is selected if and only if the expression yields true.

As the expression is composed of side-effect free components, the order of evaluation does not need to be (and is not) defined. Similarly, for conjunction (&&) and disjunction (||) (defined later), both a short-circuiting and a fully evaluating implementation will lead to the same result; both implementation strategies are therefore valid.

The current node is accessible via the current node identifier @. This identifier addresses the current node of the filter-selector that is directly enclosing the identifier; note that within nested filter-selectors, there is no syntax to address the current node of any other than the directly enclosing filter-selector (i.e., of filter-selectors enclosing the filter-selector that is directly enclosing the identifier).

A test expression either tests the existence of a node designated by an embedded query (see Section "Existence Tests") or tests the
result of a function expression (see Section 2.6). In the latter case, if the function expression is of type OptionalBoolean or one of its subtypes, it tests whether the result is true; if the function expression is of type OptionalNodes or one of its subtypes, it tests whether the result is different from Nothing.

Parentheses MAY be used within boolean-expr for grouping.

Comparisons are between primitive values (that is, numbers, strings, true, false, and null), Singular Paths, each of which selects at most one node, and function expressions (see Section 2.6) that return a primitive value or at most one node.
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).

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

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Grouping</td>
<td>(...)</td>
</tr>
<tr>
<td>4</td>
<td>Logical NOT</td>
<td>!</td>
</tr>
<tr>
<td>3</td>
<td>Relations</td>
<td>== !&gt;=</td>
</tr>
<tr>
<td>2</td>
<td>Logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>1</td>
<td>Logical OR</td>
<td></td>
</tr>
</tbody>
</table>

Table 10: 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 primitive values, it selects nothing.

The order in which the children of an object appear in the resultant nodelist is not stipulated, since JSON objects are unordered. Children of an array appear in array order in the resultant nodelist.

**Existence Tests**

A path by itself in a Boolean context is an existence test which yields true if the path selects at least one node and yields false if the path does not select any nodes.

Existence tests differ from comparisons in that:

*they work with arbitrary relative or absolute paths (not just Singular Paths).

*they work with paths that select 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 2.8) rather than the negated existence test !@.foo (which yields false if @.foo selects a node, regardless of the node's value).

**Comparisons**

The comparison operators == and < are defined first and then these are used to define !=, <=, >, and >=.

When a path resulting in an empty nodelist appears on either side of a comparison:

*a comparison using the operator == yields true if and only if the comparison is between two paths each of which result in an empty nodelist.

*a comparison using the operator < yields false.
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:

- numbers expected to interoperate as per Section 2.2 of I-JSON [RFC7493] that compare equal using normal mathematical equality,

- numbers at least one of which is not expected to interoperate as per I-JSON, where the numbers compare equal using an implementation specific equality,

- equal primitive values which are not numbers,

- equal arrays, that is arrays of the same length where each element of the first array is equal to the corresponding element of the second array, or

- equal objects with no duplicate names, that is where:

  o both objects have the same collection of names (with no duplicates), and

  o for each of those names, the values associated with the name by the objects are equal.

*a comparison using the operator < yields true if and only if the comparison is between values which are both numbers or both strings and which satisfy the comparison:

- numbers expected to interoperate as per Section 2.2 of I-JSON [RFC7493] MUST compare using the normal mathematical ordering; numbers not expected to interoperate as per I-JSON 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 the operator < yield false if either value being compared is an object, array, boolean, or null.
!=, <=, >, and => are defined in terms of the other comparison operators. For any a and b:

*The comparison a != b yields true if and only if a == b yields false.

*The comparison a <= b yields true if and only if a < b yields true or a == b yields true.

*The comparison a > b yields true if and only if b < a yields true.

*The comparison a >= b yields true if and only if b < a yields true or a == b yields true.

Boolean Operators

The logical AND, OR, and NOT operators have the normal semantics of Boolean algebra and obey its laws (see, for example, [BOOLEAN-LAWS]).

Function Extensions

Filter selectors may use function extensions, which are covered in Section 2.6.

Examples

The first set of examples shows some comparison expressions and their result with a given JSON value as input.

JSON:

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

Comparisons:

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.absent1 == $.absent2</td>
<td>true</td>
<td>Empty nodelist</td>
</tr>
<tr>
<td>$.absent1 &lt;= $.absent2</td>
<td>true</td>
<td>== implies &lt;=</td>
</tr>
<tr>
<td>$.absent == 'g'</td>
<td>false</td>
<td>Empty nodelist</td>
</tr>
<tr>
<td>$.absent1 != $.absent2</td>
<td>false</td>
<td>Empty nodelists</td>
</tr>
<tr>
<td>$.absent != 'g'</td>
<td>true</td>
<td>Empty nodelist</td>
</tr>
<tr>
<td>1 &lt;= 2</td>
<td>true</td>
<td>Numeric comparison</td>
</tr>
<tr>
<td>1 &gt; 2</td>
<td>false</td>
<td>Strict, numeric comparison</td>
</tr>
<tr>
<td>13 == '13'</td>
<td>false</td>
<td>Type mismatch</td>
</tr>
</tbody>
</table>
Comparison Result Comment
'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 true == implies <=
$.arr <= $.arr true == implies <=
1 <= $.arr false Arrays are not ordered
1 > $.arr false Arrays are not ordered
$.arr <= $.arr false Objects and arrays are not ordered
true <= true true == implies <=
true > true false Booleans are not ordered

Table 11: Comparison examples

The second set of examples shows some complete JSONPath queries that make use of filter selectors, and the results of evaluating these queries on a given JSON value as input. (Note that two of the queries employ function extensions; please see Sections 2.6.5 and 2.6.6 below for details about these.)

JSON:

```
{
  "a": [3, 5, 1, 2, 4, 6, {"b": "j"}, {"b": "k"},
        {"b": []}, {"b": "kilo"}],
  "o": {"p": 1, "q": 2, "r": 3, "s": 5, "t": {"u": 6}},
  "e": "f"
}
```

Queries:

The following examples show the filter selector in use by a child segment.

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Result Paths</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.a[?@.b == 'kilo']</td>
<td>{&quot;b&quot;: &quot;kilo&quot;}</td>
<td>$['a'][9]</td>
<td>Member value comparison</td>
</tr>
<tr>
<td>$.a[?@&gt;3.5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td>Result Paths</td>
<td>Comment</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>------------------------------</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$['a']$</td>
<td>Array value comparison</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\cdot a[?@.b]$</td>
<td>$['a']$</td>
<td>Array value existence</td>
<td></td>
</tr>
<tr>
<td>[3, 5, 1, 2, 4, 6,</td>
<td>$['a']$</td>
<td>Existence of non-singular paths</td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: &quot;j&quot;},</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: &quot;k&quot;},</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: {}}</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: &quot;kilo&quot;}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[?@.*]$</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3, 5, 1, 2, 4, 6,</td>
<td>$['a']$</td>
<td>Nested filters</td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: &quot;j&quot;},</td>
<td>$['o']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: &quot;k&quot;},</td>
<td>$['o']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: {}}</td>
<td>$['o']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;b&quot;: &quot;kilo&quot;}</td>
<td>$['o']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[?@[?@.b]]$</td>
<td>$['a']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\cdot o[?@&lt;3, ? @&lt;3]$</td>
<td>$['a']$</td>
<td>Non-deterministic ordering</td>
<td></td>
</tr>
<tr>
<td>$\cdot a[?@&lt;2</td>
<td></td>
<td>@.b == &quot;k&quot;]$</td>
<td>$['a']$</td>
</tr>
<tr>
<td>$\cdot a[? match(@.b, &quot;[jk&quot;])]$</td>
<td>$['a']$</td>
<td>Array value regular expression match</td>
<td></td>
</tr>
<tr>
<td>$\cdot a[? search(@.b, &quot;[jk&quot;])]$</td>
<td>$['a']$</td>
<td>Array value regular expression search</td>
<td></td>
</tr>
</tbody>
</table>
Table 12: Filter selector examples

The example above with the query $.o[?@<3, ?@<3] shows that a filter selector may produce nodelists in distinct orders each time it appears in the child segment.
2.6. Function Extensions

Beyond the filter expression functionality defined in the preceding subsections, JSONPath defines an extension point that can be used to add filter expression functionality: "Function Extensions".

This section defines the extension point as well as four function extensions that use this extension point. While these mechanisms are designed to use the extension point, they are an integral part of the JSONPath specification and are mandatory to implement.

A function extension defines a registered name (see Section 3.2) that can be applied to a sequence of zero or more arguments, producing a result.

A function extension MUST be defined such that its evaluation is side-effect free, i.e., all possible orders of evaluation and choices of short-circuiting or full evaluation of an expression containing it must lead to the same result. (Note that memoization or logging are not side effects in this sense as they are visible at the implementation level only – they do not influence the result of the evaluation.)

A function argument is a filter-path or a comparable. According to Section 2.5.5, a function-expr is valid as a filter-path or a comparable.

Any function expressions in a query must be well-formed (by conforming to the above ABNF) and correctly typed, otherwise the JSONPath implementation MUST raise an error (see Section 2.1). To define which function expressions are correctly typed, a type system is first introduced.

2.6.1. Type System for Function Expressions

Each argument and result of a function extension must have a declared type.
A type is a set of instances. A type is a subtype of another type if its set of instances (possibly after coercion) is a subset of the set of instances of the other type.

Table 13 defines the available types in terms of abstract instances, where n denotes a node, v denotes a value, and nl denotes a non-empty nodelist. The table also lists the subtypes of each type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Abstract Instances</th>
<th>Subtypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptionalNodeOrValue</td>
<td>Node(n), Value(v), Nothing</td>
<td>OptionalNode, OptionalValue, Value</td>
</tr>
<tr>
<td>OptionalNode</td>
<td>Node(n), Nothing</td>
<td>Value, OptionalBoolean</td>
</tr>
<tr>
<td>OptionalValue</td>
<td>Value(v), Nothing</td>
<td>Value, OptionalBoolean</td>
</tr>
<tr>
<td>OptionalBoolean</td>
<td>Value(true), Value(false), Nothing</td>
<td>Boolean</td>
</tr>
<tr>
<td>Boolean</td>
<td>Value(true), Value(false)</td>
<td></td>
</tr>
<tr>
<td>OptionalNodes</td>
<td>Nodes(nl), Nothing</td>
<td>OptionalNode</td>
</tr>
</tbody>
</table>

Table 13: Function extension type system

Notes:

*OptionalNodeOrValue is an abstraction of a comparable (which may appear on either side of a comparison or as a function argument).

*OptionalNode is an abstraction of a Singular Path.

*Value is an abstraction of a primitive value.

*Boolean is an abstraction of a primitive value that is either true or false.

*OptionalValue is an abstraction of a primitive value that may alternatively be absent (Nothing).

*OptionalNodes is an abstraction of a filter-path (which appears in a test expression or as a function argument).

The abstract instances above can be obtained from the concrete representations in Table 14.

<table>
<thead>
<tr>
<th>Abstract Instance</th>
<th>Concrete Representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node(n)</td>
<td>Singular Path resulting in a nodelist containing just the node n</td>
</tr>
<tr>
<td>Value(v)</td>
<td>JSON value v</td>
</tr>
<tr>
<td>Nothing</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Instance | Concrete Representations
--- | ---
Singular Path or filter-path resulting in an empty nodelist | Nodes(n)
filtar-path resulting in the non-empty nodelist n1

Table 14: Concrete representations of abstract instances

The following subtype relationships depend on coercion:

*OptionalNode is a subtype of OptionalValue via coercion since the OptionalNode instance Node(n) can be coerced to the OptionalValue instance Value(v), where v is the value of the node n.

*OptionalNode is a subtype of OptionalNodes via coercion since the OptionalNode instance Node(n) can be coerced to the OptionalNodes instance Nodes(l), where l is a nodelist consisting of just the node n.

The type correctness of function expressions can now be defined in terms of this type system.

2.6.2. Type Correctness of Function Expressions

A function expression is correctly typed if all the following are true:

*If it occurs as a filter-path in a test expression, the function is defined to have result type OptionalNodes or one of its subtypes, or to have result type OptionalBoolean or one of its subtypes.

*If it occurs as a comparable in a comparison, the function is defined to have result type OptionalNodeOrValue or one of its subtypes.

*For it and any function expression it contains, each argument of the function matches the defined type of the argument according to one of the following rules:

- The argument is a function expression with defined result type that is the same as, or a subtype of, the defined type of the argument.

- The argument is a literal primitive value and the defined type of the argument is Value or any type of which Value is a subtype.
- The argument is a Singular Path and the defined type of the argument is OptionalNode or any type of which OptionalNode is a subtype.

- The argument is a filter-path or a Singular Path and the defined type of the argument is OptionalNodes.

### 2.6.3. length Function Extension

**Arguments**: OptionalValue

**Result**: OptionalValue (unsigned integer or Nothing)

The "length" function extension provides a way to compute the length of a value and make that available for further processing in the filter expression:

```bash
$[?length(@.authors) >= 5]
```

Its only argument is an optional value (possibly taken from a singular path as in the example above). The result also is an optional value, an unsigned integer.

*If the argument value is a string, the result is the number of Unicode scalar values in the string.

*If the argument value is an array, the result is the number of elements in the array.

*If the argument value is an object, the result is the number of members in the object.

*For any other argument value, the result is Nothing.

### 2.6.4. count Function Extension

**Arguments**: OptionalNodes

**Result**: Value (unsigned integer)

The "count" function extension provides a way to obtain the number of nodes in a nodelist and make that available for further processing in the filter expression:

```bash
$[?count(@.*.author) >= 5]
```

Its only argument is a nodelist. The result is a value, an unsigned integer, that gives the number of nodes in the nodelist. Note that there is no deduplication of the nodelist.
2.6.5. match Function Extension

Arguments:  OptionalNodeOrValue (string)

2. Value (string conforming to [I-D.draft-ietf-jsonpath-iregexp])

Result:  OptionalBoolean (true, false, or Nothing)

The "match" function extension provides a way to check whether (the entirety of, see Section 2.6.6 below) a given string matches a given regular expression, which is in [I-D.draft-ietf-jsonpath-iregexp] form.

$[?match(@.date, "1974-05-..")]

Its first argument is an optional string that is matched against the iregexp contained in the string that is the second argument. The result is true if the string matches the iregexp and false otherwise.

The result is Nothing if the first argument is not a string or the second argument is not a string conforming to [I-D.draft-ietf-jsonpath-iregexp].

2.6.6. search Function Extension

Arguments:  OptionalNodeOrValue (string)

2. Value (string conforming to [I-D.draft-ietf-jsonpath-iregexp])

Result:  OptionalBoolean (true, false, or Nothing)

The "search" function extension provides a way to check whether a given string contains a substring that matches a given regular expression, which is in [I-D.draft-ietf-jsonpath-iregexp] form.

$[?search(@.author, "[BR]ob")]

Its first argument is an optional string that is searched for at least one substring that matches the iregexp contained in the string that is the second argument. The result is true if such a substring exists, false otherwise.

The result is Nothing if the first argument is not a string or the second argument is not a string conforming to [I-D.draft-ietf-jsonpath-iregexp].
Examples

<table>
<thead>
<tr>
<th>Query</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[?length(@) &lt; 3]</td>
<td>Valid typing</td>
</tr>
<tr>
<td>$[?length(@.*) &lt; 3]</td>
<td>Invalid typing since @.* is a non-singular path</td>
</tr>
<tr>
<td>$[?count(@.*) == 1]</td>
<td>Valid typing</td>
</tr>
<tr>
<td>$[?count(1) == 1]</td>
<td>Invalid typing since 1 is not a path</td>
</tr>
<tr>
<td>$[?count(foo(@.*))) == 1]</td>
<td>Valid typing, where foo is a function extension with argument of type OptionalNodes and result type OptionalNodes</td>
</tr>
<tr>
<td>$[?match(@.timezone, 'Europe/.*')]</td>
<td>Valid typing</td>
</tr>
<tr>
<td>$[?match(@.timezone, 'Europe/.*')] == true]</td>
<td>Invalid typing since OptionBoolean is not a comparable</td>
</tr>
</tbody>
</table>

Table 15: Function expression examples

2.7. Segments

Segments apply one or more selectors to an input value and concatenate the results into a single nodelist.

It turns out that the more segments there are in a query, the greater the depth in the input value of the nodes of the resultant nodelist:

*A query with N segments, where N >= 0, produces a nodelist consisting of nodes at depth in the input value of N or greater.

*A query with N segments, where N >= 0, all of which are child segments (Section 2.7.1), produces a nodelist consisting of nodes precisely at depth N in the input value.

There are two kinds of segment: child segments and descendant segments.

segment = child-segment / descendant-segment

The syntax and semantics of each kind of segment are defined below.

2.7.1. Child Segment

Syntax

The child segment consists of a non-empty, comma-separated sequence of selectors enclosed in square brackets.
Shorthand notations are also provided for when there is a single wildcard or name selector.

child-segment = bracketed-selection / 
    ("." 
    (wildcard-selector / 
    member-name-shorthand))

bracketed-selection = "[" S selector *(S "," S selector) S "]"

member-name-shorthand = 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

`.`, a child-segment directly built from a wildcard-selector, is shorthand for `[*]`.

`<member-name>`, a child-segment built from a member-name-shorthand, is shorthand for `['<member-name>']`. Note that this can only be used with member names that are composed of certain characters, as specified in the ABNF rule member-name-shorthand. Thus, for example, `.foo.bar` is shorthand for `$('foo')['bar']` (but not for `$['foo.bar']`).

**Semantics**

A child segment contains a sequence of selectors, each of which selects zero or more children of the input value.

Selectors of different kinds may be combined within a single child segment.

The resulting nodelist of a child segment is the concatenation of the nodelists from each of its selectors in the order that the selectors appear in the list. Note that any node matched by more than one selector is kept as many times in the nodelist.

Where a selector can produce a nodelist in more than one possible order, each occurrence of the selector in the child segment may evaluate to produce a nodelist in a distinct order.

So a child segment drills down one more level into the structure of the input value.
Examples

JSON:

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

Queries:

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Result Paths</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[0, 3]</code></td>
<td>&quot;a&quot;</td>
<td><code>[0]</code></td>
<td>Indices</td>
</tr>
<tr>
<td></td>
<td>&quot;d&quot;</td>
<td><code>[3]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;a&quot;</td>
<td><code>[0]</code></td>
<td></td>
</tr>
<tr>
<td><code>[0:2, 5]</code></td>
<td>&quot;b&quot;</td>
<td><code>[1]</code></td>
<td>Slice and index</td>
</tr>
<tr>
<td></td>
<td>&quot;f&quot;</td>
<td><code>[5]</code></td>
<td></td>
</tr>
<tr>
<td><code>[0, 0]</code></td>
<td>&quot;a&quot;</td>
<td><code>[0]</code></td>
<td>Duplicated entries</td>
</tr>
</tbody>
</table>

Table 16: Child segment examples

2.7.2. Descendant Segment

Syntax

The descendant segment consists of a double dot .. followed by a child segment (using bracket notation).

Shortand notations are also provided that correspond to the shorthand forms of the child segment.

descendant-segment = ".." (bracketed-selection / wildcard-selector / member-name-shorthand)

..*, the descendant-segment directly built from a wildcard-selector, is shorthand for ..[*].

..<member-name>, a descendant-segment built from a member-name-shorthand, is shorthand for ..["<member-name>"]. As with the similar shorthand of a child-segment, note that this can only be used with member names that are composed of certain characters, as specified in the ABNF rule member-name-shorthand.

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

Semantics

A descendant segment produces zero or more descendants of the input value.
A descendant selector visits the input value and each of its descendants such that:

* nodes of any array are visited in array order, and
* nodes are visited before their descendants.

The order in which the children of an object are visited is not stipulated, since JSON objects are unordered.

Suppose the descendant segment is of the form ..[<selectors>] (after converting any shorthand form to bracket notation) and the nodes, in the order visited, are D1, ..., Dn (where n >= 1). Note that D1 is the input value.

For each i such that 1 <= i <= n, the nodelist Ri is defined to be a result of applying the child segment [<selectors>] to the node Di.

The result of the descendant selector is the concatenation of R1, ..., Rn (in that order).

So a descendant segment drills down one or more levels into the structure of the input value.

**Examples**

**JSON:**

```json
{
  "o": {"j": 1, "k": 2},
  "a": [5, 3, [{"j": 4}, {"k": 6}]]
}
```

**Queries:**

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Result Paths</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$..j</td>
<td>1</td>
<td>$['o'][&quot;j&quot;]</td>
<td>Object values</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>$['a'][2][0][&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td>$..j</td>
<td>4</td>
<td>$['a'][2][0][&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>$['a'][2][0]</td>
<td>Alternative result</td>
</tr>
<tr>
<td>$..[0]</td>
<td>5</td>
<td>$['o'][&quot;j&quot;]</td>
<td>Array values</td>
</tr>
<tr>
<td>$..[0]</td>
<td></td>
<td>$['a'][0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{&quot;j&quot;: 4}</td>
<td>$['a'][2][0]</td>
<td></td>
</tr>
<tr>
<td>$..[0]</td>
<td></td>
<td>$['a'][0]</td>
<td>Alternative result</td>
</tr>
<tr>
<td></td>
<td>{&quot;j&quot;: 4}</td>
<td>$['a'][2][0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>$['a'][0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{&quot;j&quot;: 1, &quot;k&quot;: 2}</td>
<td>$['o']</td>
<td></td>
</tr>
<tr>
<td>$..[*]</td>
<td>[5, 3, [{&quot;j&quot;: 4}, {&quot;k&quot;: 6}]]</td>
<td>$['o'][&quot;j&quot;]</td>
<td>All values</td>
</tr>
<tr>
<td>$..*</td>
<td></td>
<td>$['a']</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$['o'][&quot;j&quot;]</td>
<td></td>
</tr>
<tr>
<td>Query</td>
<td>Result</td>
<td>Result Paths</td>
<td>Comment</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>--------------</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>$['o']['k']$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$['a'][0]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$['a'][1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$['a'][2]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[&quot;j&quot;: 4], [&quot;k&quot;: 6]</td>
<td>$['a'][2][0]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$['a'][2][0]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$['a'][2][1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$['a'][2][1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$['a'][2][1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$['a'][2][1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$['a'][2][1]$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$['a'][2][1]$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$.o..[*, *, *]$ | $['o']['j']$ |         |         |

$.o..[*] | $['o']['k']$ |         |         |

$.o..[*] | $['o']['j']$ | Non-deterministic ordering |         |

$.a..[0, 1] | $['a'][0][2]$ | Multiple segments |         |

Table 17: Descendant segment examples

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

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

* [5, 3, "{"j": 4}, "{k": 6}] must appear before 5, 3, and [{"j": 4}, "{k": 6}],

* 5 must appear before 3 which must appear before [{"j": 4}, "{k": 6}],

* 5 and 3 must appear before "{j": 4}, 4, , "{k": 6}, and 6,

* [{"j": 4}, "{k": 6}] must appear before "{j": 4} and "{k": 6},

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

* "{k": 6} must appear before 6.

The example above with the query $.o..[*] shows that a selector may produce nodelists in distinct orders each time it appears in the descendant segment.
The example above with the query $.a..[0, 1] shows that the child segment [0, 1] is applied to each node in turn (rather than the nodes being visited once per selector, which is the case for some JSONPath implementations that do not conform to this specification).

2.8. 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:

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
<th>Result Paths</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.a</td>
<td>null</td>
<td>$['a']</td>
<td>Object value</td>
</tr>
<tr>
<td>$.a[0]</td>
<td>null</td>
<td></td>
<td>null used as array</td>
</tr>
<tr>
<td>$.a.d</td>
<td></td>
<td></td>
<td>null used as object</td>
</tr>
<tr>
<td>$.b[0]</td>
<td>null</td>
<td>$['b'][0]</td>
<td>Array value</td>
</tr>
<tr>
<td>$.b[*]</td>
<td>null</td>
<td>$['b'][0]</td>
<td>Array value</td>
</tr>
<tr>
<td>$.b[?@]</td>
<td>null</td>
<td>$['b'][0]</td>
<td>Existence</td>
</tr>
<tr>
<td>$.b[?@==null]</td>
<td>null</td>
<td>$['b'][0]</td>
<td>Comparison</td>
</tr>
<tr>
<td>$.c[?(@.d==null)]</td>
<td></td>
<td></td>
<td>Comparison with &quot;missing&quot; value</td>
</tr>
<tr>
<td>$.null</td>
<td>1</td>
<td>$['null']</td>
<td>Not JSON null at all, just a member name string</td>
</tr>
</tbody>
</table>

Table 18: Examples involving (or not involving) null

2.9. Normalized Paths

A Normalized Path is a canonical representation of the location of a node in a value and uniquely identifies the node in the value. Specifically, a Normalized Path is a JSONPath query with restricted syntax (defined below), e.g., $['book'][3], which when applied to the value results in a nodelist consisting of just the node identified by the Normalized Path. Note that a Normalized Path represents the identity of a node in a specific value. There is precisely one Normalized Path identifying any particular node in a value.

A canonical representation of a nodelist is as a JSON arrays of strings, where the strings are Normalized Paths.
Normalized Paths provide a predictable format that simplifies testing and post-processing of nodelists, e.g., to remove duplicate nodes. Normalized Paths are used in this document as result paths in examples.

Normalized Paths use the canonical bracket notation, rather than dot notation.

Single quotes are used to delimit string member names. This reduces the number of characters that need escaping when Normalized Paths appear in double quote delimited strings, e.g., in JSON texts.

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

Note: Normalized Paths are Singular Paths, but not all Singular Paths are Normalized Paths. For example, $[-3]$ is a Singular Path, but is not a Normalized Path. The Normalized Path equivalent to $[-3]$ would have an index equal to the array length minus 3. (The array length must be at least 3 if $[-3]$ is to identify a node.)
Since there can only be one Normalized Path identifying a given node, the syntax stipulates which characters are escaped and which are not. So the definition of normal-hexchar is designed for hex escaping of characters which are not straightforwardly-printable, for example U+000B LINE TABULATION, but for which no standard JSON escape, such as \\n, is available.

Examples

<table>
<thead>
<tr>
<th>Path</th>
<th>Normalized Path</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.a</td>
<td>$['a']</td>
<td>Object value</td>
</tr>
<tr>
<td>$[1]</td>
<td>$[1]</td>
<td>Array index</td>
</tr>
<tr>
<td>$[-3]</td>
<td>$[2]</td>
<td></td>
</tr>
</tbody>
</table>
3. IANA Considerations

3.1. 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

### 3.2. Function Extensions

This specification defines a new "Function Extensions sub-registry" in a new "JSONPath Parameters registry", with the policy "expert review" ([Section 4.5](BCP26)).

The experts are instructed to be frugal in the allocation of function extension names that are suggestive of generally applicable semantics, keeping them in reserve for functions that are likely to enjoy wide use and can make good use of their conciseness. The expert is also instructed to direct the registrant to provide a specification ([Section 4.6](BCP26)), but can make exceptions, for instance when a specification is not available at the time of registration but is likely forthcoming. If the expert becomes aware of function extensions that are deployed and in use, they may also initiate a registration on their own if they deem such a registration can avert potential future collisions.

Each entry in the registry must include:

**Function Name:**
- A lower case ASCII [STD80] string that starts with a letter and can contain letters, digits and underscore characters afterwards ([a-z][_a-z0-9]*)

**Brief description:**
- A brief description

**Input:**
- A comma-separated list of zero or more types of the arguments expected for this function extension

**Output:**
- The type of the result for this function extension

**Change Controller:**
- (see [Section 2.3](BCP26))
Reference:

a reference document that provides a description of the function extension

Initial entries in this sub-registry are as listed in Table 20; the Column "Change Controller" always has the value "IESG" and the column "Reference" always has the value "Section 2.6 of RFCthis":

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Brief description</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>length of array</td>
<td>OptionalValue</td>
<td>OptionalValue</td>
</tr>
<tr>
<td>count</td>
<td>size of nodelist</td>
<td>OptionalNodes</td>
<td>Value</td>
</tr>
<tr>
<td>match</td>
<td>regular expression full match</td>
<td>OptionalNodeOrValue, Value</td>
<td>OptionalBoolean</td>
</tr>
<tr>
<td>search</td>
<td>substring match</td>
<td>OptionalNodeOrValue, Value</td>
<td>OptionalBoolean</td>
</tr>
</tbody>
</table>

Table 20: Initial Entries in the Function Extensions Subregistry

4. Security Considerations

Security considerations for JSONPath can stem from

* attack vectors on JSONPath implementations,

* attack vectors on how JSONPath queries are formed, and

* the way JSONPath is used in security-relevant mechanisms.

4.1. 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's eval() function. 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-ietf-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 segment, stack overflow. Implementations need to have appropriate resource management to mitigate these attacks.

4.2. Attack Vectors on How JSONPath Queries are Formed

JSONPath queries are often not static, but formed from variables that provide index values, member names, or values to compare with in a filter expression. These variables need to be translated into the form they take in a JSONPath query, e.g., by escaping string delimiters, or by only allowing specific constructs such as .name to be formed when the given values allow that. Failure to perform these translations correctly can lead to unexpected failures, which can lead to Availability, Confidentiality, and Integrity breaches, in particular if an adversary has control over the values (e.g., by entering them into a Web form). The resulting class of attacks, injections (e.g., SQL injections), is consistently found among the top causes of application security vulnerabilities and requires particular attention.

4.3. 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.

5. References

5.1. Normative References

5.2. Informative References
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

\[/\text{store}/\text{book}[1]/\text{title}\]

can be realized in the expression

\[\text{x.store.book}[0].\text{title}\]

or, in bracket notation,

\[\text{x['store']['book'][0]['title']}\]

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 (\$.\text{store}.\text{book}[0].\text{title}) or the bracket notation ($['\text{store}']['\text{book}'][0]['\text{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 ECMASCRIP 4.

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

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

Table 21 extends Table 1 by providing a comparison with similar XPath concepts.

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

Table 21: XPath syntax compared to JSONPath

For further illustration, Table 22 shows some XPath expressions and their JSONPath equivalents.

<table>
<thead>
<tr>
<th>XPath</th>
<th>JSONPath</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>/store/book/author</td>
<td>$.store.book[*].author</td>
<td>the authors of all books in the store</td>
</tr>
<tr>
<td>//author</td>
<td>$.author</td>
<td>all authors</td>
</tr>
<tr>
<td>/store/*</td>
<td>$.store.*</td>
<td></td>
</tr>
<tr>
<td>XPath</td>
<td>JSONPath</td>
<td>Result</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>/store//price</td>
<td>$.store..price</td>
<td>all things in store, which are some books and a red bicycle</td>
</tr>
<tr>
<td>//*</td>
<td>$..*</td>
<td>filter all books cheaper than 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all elements in XML document; all member values and array elements contained in input value</td>
</tr>
</tbody>
</table>

Table 22: 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 each of the nodes in the nodelist resulting from 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 value of an object or an element of an array. For conversion to a JSONPath query, knowledge of the structure of the JSON value is needed to distinguish these cases.

Acknowledgements

This document 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

Greg Dennis
Auckland
New Zealand

Email: gregsdennis@yahoo.com
URI: https://github.com/gregsdennis

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