

Workgroup: JSONPath WG  
Internet-Draft: draft-ietf-jsonpath-base-13  
Published: 15 April 2023  
Intended Status: Standards Track  
Expires: 17 October 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 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 17 October 2023.

## Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

- [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](#)
    - [2.1.1. Syntax](#)
    - [2.1.2. Semantics](#)
    - [2.1.3. Example](#)
  - [2.2. Root Identifier](#)
  - [2.3. Selectors](#)
    - [2.3.1. Name Selector](#)
    - [2.3.2. Wildcard Selector](#)
    - [2.3.3. Index Selector](#)
      - [2.3.3.1. Syntax](#)
      - [2.3.3.2. Semantics](#)
      - [2.3.3.3. Examples](#)
    - [2.3.4. Array Slice selector](#)
      - [2.3.4.1. Syntax](#)
      - [2.3.4.2. Semantics](#)
      - [2.3.4.3. Examples](#)
    - [2.3.5. Filter selector](#)
      - [2.3.5.1. Syntax](#)
      - [2.3.5.2. Semantics](#)
      - [2.3.5.3. Examples](#)

- [2.4. Function Extensions](#)
  - [2.4.1. Type System for Function Expressions](#)
  - [2.4.2. Type Conversion](#)
  - [2.4.3. Well-Typedness of Function Expressions](#)
  - [2.4.4. length\(\) Function Extension](#)
  - [2.4.5. count\(\) Function Extension](#)
  - [2.4.6. match\(\) Function Extension](#)
  - [2.4.7. search\(\) Function Extension](#)
  - [2.4.8. value\(\) Function Extension](#)
  - [2.4.9. Examples](#)
- [2.5. Segments](#)
  - [2.5.1. Child Segment](#)
  - [2.5.2. Descendant Segment](#)
- [2.6. Semantics of null](#)
- [2.7. Normalized Paths](#)
- [3. IANA Considerations](#)
  - [3.1. Registration of Media Type application/jsonpath](#)
  - [3.2. Function Extensions](#)
- [4. Security Considerations](#)
  - [4.1. Attack Vectors on JSONPath Implementations](#)
  - [4.2. Attack Vectors on How JSONPath Queries are Formed](#)
  - [4.3. Attacks on Security Mechanisms that Employ JSONPath](#)
- [5. References](#)
  - [5.1. Normative References](#)
  - [5.2. Informative References](#)
- [Appendix A. Inspired by XPath](#)
  - [A.1. JSONPath and XPath](#)
- [Appendix B. JSON Pointer](#)
- [Acknowledgements](#)
- [Contributors](#)
- [Authors' Addresses](#)

## **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.

Functions are referred to using the function name followed by a pair of parentheses, as in `fname()`.

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 constituents 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.

**Query Argument:** Short name for the value a JSONPath expression is applied to. (Also used for actual parameters of function-expressions.)

**Location:** the position of a value within the query argument. This can be thought of as a sequence of names and indexes navigating to the value through the objects and arrays in the query argument, with the empty sequence indicating the query 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 query argument.

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

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

**Parameter:** Formal parameter (of a function) that can take a function argument (an actual parameter) in a function-expression.

**Normalized Path:** A form of JSONPath expression that identifies a node in a value by providing a query that results in exactly that node. Each node in a query argument is identified by exactly one Normalized Path (we say, the Normalized Path is "unique" for that node), and, to be a Normalized Path for a specific query argument, the Normalized Path needs to identify exactly one 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 Query:** A JSONPath expression built from segments each of which, regardless of the input value, produces a nodelist containing at most one node.

**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 query 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 query argument as a nodelist (a list of nodes).

Nodes are the selectable parts of the query 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 is based on 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.

The term *JSONPath* was chosen because of the XPath inspiration and also because the outcome of a query consists of *paths* identifying nodes in the JSON query argument.

### 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 [2.3.1](#), [2.3.2](#), [2.3.5](#), and [2.5.2](#) 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 2.3.1](#)) and comparing strings ([Section 2.3.5.2.2](#) in [Section 2.3.5](#)) 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 query 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 query 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 ([Section 2.3.5](#)).

#### 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 2.3.2](#)) 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 2.3.4](#)) 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 `?<logical-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.

Syntax Element	Description
\$	<a href="#">root node identifier</a> ( <a href="#">Section 2.2</a> )
@	<a href="#">current node identifier</a> ( <a href="#">Section 2.3.5</a> ) (valid only within filter selectors)
[<selectors>]	<a href="#">child segment</a> ( <a href="#">Section 2.5.1</a> ) selects zero or more children of a node; contains one or more selectors, separated by commas
.name	shorthand for ['name']
.*	shorthand for [*]
.. [<selectors>]	<a href="#">descendant segment</a> ( <a href="#">Section 2.5.2</a> ): selects zero or more descendants of a node; contains one or more selectors, separated by commas
..name	shorthand for ..['name']
..*	shorthand for ..[*]
'name'	<a href="#">name selector</a> ( <a href="#">Section 2.3.1</a> ): selects a named child of an object
*	<a href="#">wildcard selector</a> ( <a href="#">Section 2.3.1</a> ): selects all children of a node
3	<a href="#">index selector</a> ( <a href="#">Section 2.3.3</a> ): selects an indexed child of an array (from 0)
0:100:5	<a href="#">array slice selector</a> ( <a href="#">Section 2.3.4</a> ): start:end:step for arrays
?<logical-expr>	<a href="#">filter selector</a> ( <a href="#">Section 2.3.5</a> ): selects particular children using a logical expression
length(@.foo)	<a href="#">function extension</a> ( <a href="#">Section 2.4</a> ): 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).

```

{ "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.

JSONPath	Intended result
<code>\$.store.book[*].author</code>	the authors of all books in the store
<code>\$..author</code>	all authors
<code>\$.store.*</code>	all things in store, which are some books and a red bicycle
<code>\$.store..price</code>	the prices of everything in the store
<code>\$..book[2]</code>	the third book
<code>\$..book[-1]</code>	the last book in order
<code>\$..book[0,1]</code>	the first two books
<code>\$..book[:2]</code>	
<code>\$..book[?(@.isbn)]</code>	all books with an ISBN number

JSONPath	Intended result
<code>\$.book[?(@.price&lt;10)]</code>	all books cheaper than 10
<code>\$..*</code>	all member values and array elements contained in the input value

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 *query 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 *well typed*, as described in [Section 2.4](#).

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.)

### 2.1.1. Syntax

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

```
jsonpath-query      = root-identifier segments
segments            = *(S segment)
```

The syntax and semantics of segments are defined in [Section 2.5](#).

### 2.1.2. 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 *query 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 nodelists.

The nodelist resulting from the root identifier contains a single node, the query 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: 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 in the order of the input nodelist they were derived from 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.

### 2.1.3. Example

Consider this example. With the query 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`, `[*]`, and `.b`.

Firstly, `$` produces a nodelist consisting of just the query 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.

## 2.2. Root Identifier

### Syntax

Every JSONPath query (except those inside filter expressions, see [Section 2.3.5](#)) **MUST** begin with the root identifier `$`.

root-identifier = "\$"

## Semantics

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

## Examples

JSON:

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

Queries:

Query	Result	Result Path	Comment
<code>\$</code>	<code>{"k": "v"}</code>	<code>\$</code>	Root node

Table 3: Root identifier examples

### 2.3. Selectors

Selectors appear only inside [child segments](#) ([Section 2.5.1](#)) and [descendant segments](#) ([Section 2.5.2](#)).

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.

```
selector          = name-selector /  
                   wildcard-selector /  
                   slice-selector /  
                   index-selector /  
                   filter-selector
```

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

#### 2.3.1. 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
                   ; omit 0x22 "
                   %x23-26 /
                   ; omit 0x27 '
                   %x28-5B /
                   ; omit 0x5C \
                   %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

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"

```

Note: double-quoted strings follow the JSON string syntax ([Section 7](#) of [RFC8259](#)); single-quoted strings follow an analogous pattern ([Section 2.3.3.1](#)). No attempt was made to improve on this syntax, so if it is desired to escape characters with scalar values above 0xFFFF, such as U+1F914 (" ", THINKING FACE), they 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 shown in [Table 4](#):

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 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: 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 examples in [Table 5](#) show the name selector in use by child segments:



Query	Result	Result Paths	Comment
<code>\$.o['j j'] ['k.k']</code>	3	<code>\$['o']['j j'] ['k.k']</code>	Named value in nested object
<code>\$.o["j j"] ["k.k"]</code>	3	<code>\$['o']['j j'] ['k.k']</code>	Named value in nested object
<code>\$["'"]["@"]</code>	2	<code>\$['\']['@']</code>	Unusual member names

Table 5: Name selector examples

### 2.3.2. 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 examples in [Table 6](#) show the wildcard selector in use by a child segment:

Query	Result	Result Paths	Comment
<code>\$[*]</code>	<code>{"j": 1, "k": 2} [5, 3]</code>	<code>\$['o'] \$['a']</code>	Object values
<code>\$.o[*]</code>	1 2	<code>\$['o'] ['j'] \$['o']['k']</code>	Object values

Query	Result	Result Paths	Comment
<code>\$.o[*]</code>	2 1	<code>\$['o']</code> <code>['k']</code> <code>\$['o']['j']</code>	Alternative result
<code>\$.o[*, *]</code>	1 2 2 1	<code>\$['o']</code> <code>['j']</code> <code>\$['o']</code> <code>['k']</code> <code>\$['o']</code> <code>['k']</code> <code>\$['o']['j']</code>	Non-deterministic ordering
<code>\$.a[*]</code>	5 3	<code>\$['a'][0]</code> <code>\$['a'][1]</code>	Array members

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).

### 2.3.3. Index Selector

#### 2.3.3.1. 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 2.3.3.2](#)).

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

Notes:

\*An index-selector is an integer (in base 10, as in JSON numbers).

\*As in JSON numbers, the syntax does not allow octal-like integers with leading zeros such as `01` or `-01`.

### 2.3.3.2. 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.

### 2.3.3.3. Examples

JSON:

```
["a","b"]
```

Queries:

The examples in [Table 7](#) show the index selector in use by a child segment.

Query	Result	Result Paths	Comment
<code>[\$[1]]</code>	"b"	<code>[\$[1]]</code>	Element of array
<code>[\$[-2]]</code>	"a"	<code>[\$[0]]</code>	Element of array, from the end

Table 7: Index selector examples

### 2.3.4. Array Slice selector

#### 2.3.4.1. 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.

slice-selector	=	[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

The slice selector consists of three optional decimal integers separated by colons. The second colon can be omitted when the third integer is.

To be valid, the integers provided **MUST** be in the I-JSON range of exact values, see [Section 2.1](#).

#### 2.3.4.2. 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 shown in [Table 8](#):

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

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:

```

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,  $a(i)$  is the  $i+1$ th element of the array  $a$  (i.e.,  $a(0)$  is the first element,  $a(1)$  the second, and so forth).

```

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  $\text{step} = 0$ , no elements are selected and the result array is empty.

#### 2.3.4.3. Examples

JSON:

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

Queries:

The examples in [Table 9](#) show the array slice selector in use by a child segment:

Query	Result	Result Paths	Comment
<code>\$\$[1:3]</code>	"b" "c"	<code>\$\$[1]</code> <code>\$\$[2]</code>	Slice with default step
<code>\$\$[5:]</code>	"f" "g"	<code>\$\$[5]</code> <code>\$\$[6]</code>	Slice with no end index
<code>\$\$[1:5:2]</code>	"b" "d"	<code>\$\$[1]</code> <code>\$\$[3]</code>	Slice with step 2
<code>\$\$[5:1:-2]</code>	"f" "d"	<code>\$\$[5]</code> <code>\$\$[3]</code>	Slice with negative step
<code>\$\$[::-1]</code>	"g" "f" "e" "d" "c" "b" "a"	<code>\$\$[6]</code> <code>\$\$[5]</code> <code>\$\$[4]</code> <code>\$\$[3]</code> <code>\$\$[2]</code> <code>\$\$[1]</code> <code>\$\$[0]</code>	Slice in reverse order

Table 9: Array slice selector examples

### 2.3.5. Filter selector

Filter selectors are used to iterate over the elements or members of structured values, i.e., JSON arrays and objects. The structured values are identified in the nodelist offered by the child or descendant segment using the filter selector.

For each iteration (element/member), a logical expression, the *filter expression*, is evaluated which decides whether the node of the element/member is selected. (While a logical expression evaluates to what mathematically is a Boolean value, this specification uses the term *logical* to maintain a distinction from the Boolean values that JSON can represent.)

During the iteration process, the filter expression receives the node of each array element or object member value of the structured value being filtered; this element or member value is then known as the *current node*.

The current node can be used as the start of one or more JSONPath queries in subexpressions of the filter expression, notated via the current-node-identifier `@`. Each JSONPath query can be used either for testing existence of a result of the query, for obtaining a specific JSON value resulting from that query that can then be used in a comparison, or as a *function argument*.

Filter selectors may use function extensions, which are covered in [Section 2.4](#). Within the logical expression for a filter selector, function expressions can be used to operate on nodelists and values. The set of available functions is extensible, with a number of

functions predefined, see [Section 2.4](#), and the ability to register further functions provided by the Function Extensions sub-registry ([Section 3.2](#)). When a function is defined, it is given a unique name, and its return value and each of its parameters is given a *declared type*. The type system is limited in scope; its purpose is to express restrictions that, without functions, are implicit in the grammar of filter expressions. The type system also guides conversions ([Section 2.4.2](#)) that mimic the way different kinds of expressions are handled in the grammar when function expressions are not in use.

#### 2.3.5.1. Syntax

The filter selector has the form ?<logical-expr>.

filter-selector        = "?" S logical-expr

As the filter expression is composed of side-effect free constituents, 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: 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).

Logical expressions offer the usual Boolean operators (|| for OR, && for AND, and ! for NOT). They have the normal semantics of Boolean algebra and obey its laws (see, for example, [[BOOLEAN-LAWS](#)]). Parentheses **MAY** be used within logical-expr for grouping.



```

logical-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        = paren-expr /
                  comparison-expr /
                  test-expr

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

```

A test expression either tests the existence of a node designated by an embedded query (see [Section 2.3.5.2.1](#)) or tests the result of a function expression (see [Section 2.4](#)). In the latter case, if the function's declared result type is `LogicalType` (see [Section 2.4.1](#)), it tests whether the result is `LogicalTrue`; if the function's declared result type is `NodesType`, it tests whether the result is non-empty. If the function's declared result type is `ValueType`, its use in a test expression is not well typed (see [Section 2.4.3](#)).

```

test-expr        = [logical-not-op S]
                  (filter-query / ; existence/non-existence
                   function-expr) ; LogicalType or
                  ; NodesType
filter-query      = rel-query / jsonpath-query
rel-query         = current-node-identifier segments
current-node-identifier = "@"

```

Comparison expressions are available for comparisons between primitive values (that is, numbers, strings, true, false, and null). These can be obtained via literal values; singular queries, each of which selects at most one node the value of which is then used; or function expressions (see [Section 2.4](#)) of type `ValueType`.

```

comparison-expr    = comparable S comparison-op S comparable
literal            = number / string-literal /
                    true / false / null
comparable          = literal /
                    singular-query / ; singular query value
                    function-expr ; ValueType
comparison-op       = "==" / "!=" /
                    "<=" / ">=" /
                    "<" / ">"

singular-query      = rel-singular-query / abs-singular-query
rel-singular-query  = current-node-identifier singular-query-segments
abs-singular-query  = root-identifier singular-query-segments
singular-query-segments = *(S (name-segment / index-segment))
name-segment        = ("[" name-selector "]") /
                    ("." member-name-shorthand)
index-segment       = "[" index-selector "]"

```

Literals can be notated in the way that is usual for JSON (with the extension that strings can use single-quote delimiters). Alphabetic characters in ABNF are case-insensitive, so within a floating point number the ABNF expression "e" can be either the value 'e' or 'E'.

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

```

number            = (int / "-0") [ 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

```

[Table 10](#) 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 10: Filter expression operator precedence

#### 2.3.5.2. 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.

In the resultant nodelist, children of an array are ordered by their position in the array. The order in which the children of an object (as opposed to an array) appear in the resultant nodelist is not stipulated, since JSON objects are unordered.

##### 2.3.5.2.1. Existence Tests

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

Existence tests differ from comparisons in that:

- \*they work with arbitrary relative or absolute queries (not just singular queries).
- \*they work with queries that select structured values.

To examine the value of a node selected by a query, an explicit comparison is necessary. For example, to test whether the node selected by the query `@.foo` has the value null, use `@.foo == null` (see [Section 2.6](#)) rather than the negated existence test `!@.foo` (which yields false if `@.foo` selects a node, regardless of the node's value).

##### 2.3.5.2.2. Comparisons

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

When either side of a comparison results in an empty nodelist or Nothing (see [Section 2.4.1](#)):

- \*a comparison using the operator `==` yields true if and only if the other side also results in an empty nodelist or Nothing.
- \*a comparison using the operator `<` yields false.

When any query or function expression on either side of a comparison results in a nodelist consisting of a single node, that side 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.

!=, <=, >, 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.

### 2.3.5.3. 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:

Comparison	Result	Comment
\$.absent1 == \$.absent2	true	Empty nodelists
\$.absent1 <= \$.absent2	true	== implies <=
\$.absent == 'g'	false	Empty nodelist
\$.absent1 != \$.absent2	false	Empty nodelists
\$.absent != '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

Comparison	Result	Comment
<code>\$.obj &lt;= \$.arr</code>	false	Objects and arrays are not ordered
<code>\$.obj &lt; \$.arr</code>	false	Objects and arrays are not ordered
<code>\$.obj &lt;= \$.obj</code>	true	<code>==</code> implies <code>&lt;=</code>
<code>\$.arr &lt;= \$.arr</code>	true	<code>==</code> implies <code>&lt;=</code>
<code>1 &lt;= \$.arr</code>	false	Arrays are not ordered
<code>1 &gt;= \$.arr</code>	false	Arrays are not ordered
<code>1 &gt; \$.arr</code>	false	Arrays are not ordered
<code>1 &lt; \$.arr</code>	false	Arrays are not ordered
<code>true &lt;= true</code>	true	<code>==</code> implies <code>&lt;=</code>
<code>true &gt; true</code>	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: two of the queries employ function extensions; please see Sections [2.4.6](#) and [2.4.7](#) 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 examples in [Table 12](#) show the filter selector in use by a child segment:

Query	Result	Result Paths	Comment
<code>\$.a[?@.b == 'kilo']</code>	<code>{"b": "kilo"}</code>	<code>['a']</code> <code>[9]</code>	Member value comparison
<code>\$.a[?@&gt;3.5]</code>	5 4 6	<code>['a']</code> <code>[1]</code> <code>['a']</code> <code>[4]</code> <code>['a']</code> <code>[5]</code>	Array value comparison
<code>\$.a[?@.b]</code>			

Query	Result	Result Paths	Comment
	<pre> {"b": "j"} {"b": "k"} {"b": {}} {"b": "kilo"} </pre>	<pre> \$['a'] [6] \$['a'] [7] \$['a'] [8] \$['a'] [9] </pre>	Array value existence
<code>\$[?@.*]</code>	<pre> [3, 5, 1, 2, 4, 6, {"b": "j"}, {"b": "k"}, {"b": {}}, {"b": "kilo"}] {"p": 1, "q": 2, "r": 3, "s": 5, "t": {"u": 6}} </pre>	<pre> \$['a'] \$['o'] </pre>	Existence of non-singular queries
<code>\$[?@[?@.b]]</code>	<pre> [3, 5, 1, 2, 4, 6, {"b": "j"}, {"b": "k"}, {"b": {}}, {"b": "kilo"}] </pre>	<pre> \$['a'] </pre>	Nested filters
<code>\$.o[?@&lt;3, ?@&lt;3]</code>	<pre> 1 2 2 1 </pre>	<pre> \$['o'] ['p'] \$['o'] ['q'] \$['o'] ['q'] \$['o'] ['p'] </pre>	Non-deterministic ordering
<code>\$.a[?@&lt;2    @.b == "k"]</code>	<pre> 1 {"b": "k"} </pre>	<pre> \$['a'] [2] \$['a'] [7] </pre>	Array value logical OR
<code>\$.a[?match(@.b, "[jk]")]</code>	<pre> {"b": "j"} {"b": "k"} </pre>	<pre> \$['a'] [6] \$['a'] [7] </pre>	Array value regular expression match
<code>\$.a[?search(@.b, "[jk]")]</code>	<pre> {"b": "j"} {"b": "k"} {"b": "kilo"} </pre>	<pre> \$['a'] [6] \$['a'] [7] \$['a'] [9] </pre>	Array value regular expression search
<code>\$.o[?@&gt;1 &amp;&amp; @&lt;4]</code>	<pre> 2 3 </pre>	<pre> \$['o'] ['q'] \$['o'] ['r'] </pre>	Object value logical AND
			Alternative result

Query	Result	Result Paths	Comment
<code>\$.o[?@&gt;1 &amp;&amp; @&lt;4]</code>	3 2	<code>\$['o']</code> <code>['r']</code> <code>\$['o']</code> <code>['q']</code>	
<code>\$.o[?@.u    @.x]</code>	<code>{"u": 6}</code>	<code>\$['o']</code> <code>['t']</code>	Object value logical OR
<code>\$.a[?(@.b == \$.x)]</code>	3 5 1 2 4 6	<code>\$['a']</code> <code>[0]</code> <code>\$['a']</code> <code>[1]</code> <code>\$['a']</code> <code>[2]</code> <code>\$['a']</code> <code>[3]</code> <code>\$['a']</code> <code>[4]</code> <code>\$['a']</code> <code>[5]</code>	Comparison of queries with no values
<code>\$.a[?(@ == @)]</code>	3 5 1 2 4 6 <code>{"b": "j"}</code> <code>{"b": "k"}</code> <code>{"b": {}}</code> <code>{"b": "kilo"}</code>	<code>\$['a']</code> <code>[0]</code> <code>\$['a']</code> <code>[1]</code> <code>\$['a']</code> <code>[2]</code> <code>\$['a']</code> <code>[3]</code> <code>\$['a']</code> <code>[4]</code> <code>\$['a']</code> <code>[5]</code> <code>\$['a']</code> <code>[6]</code> <code>\$['a']</code> <code>[7]</code> <code>\$['a']</code> <code>[8]</code> <code>\$['a']</code> <code>[9]</code>	Comparisons of primitive and of structured values

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.



## 2.4. 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 some 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 expected to be implemented like any other integral part of this specification.

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. Each registered function name is unique.

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: 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.)

```
function-name      = function-name-first *function-name-char
function-name-first = LCALPHA
function-name-char  = function-name-first / "_" / DIGIT
LCALPHA             = %x61-7A ; "a".."z"

function-expr      = function-name "(" S [function-argument
                        *(S "," S function-argument)] S ")"
function-argument  = literal /
                        filter-query / ; (includes singular-query)
                        logical-expr /
                        function-expr
```

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

### 2.4.1. Type System for Function Expressions

Each parameter as well as the result of a function extension must have a declared type.

Declared types enable checking a JSONPath query for well-typedness independent of any query argument the JSONPath query is applied to.

[Table 13](#) defines the available types in terms of the instances they contain.

Type	Instances
ValueType	JSON values or Nothing
LogicalType	LogicalTrue or LogicalFalse
NodesType	NodeLists

Table 13: Function extension type system

Notes:

\*The only instances that can be directly represented in JSONPath syntax are certain JSON values in ValueType expressed as literals (which, in JSONPath, are limited to primitive values).

\*Nothing represents the absence of a JSON value and is distinct from any JSON value, including null.

\*LogicalTrue and LogicalFalse are unrelated to the JSON values expressed by the literals true and false.

#### 2.4.2. Type Conversion

Just as queries can be used in logical expressions by testing for the existence of at least one node ([Section 2.3.5.2.1](#)), a function expression of declared type NodesType can be used as a function argument for a parameter of declared type LogicalType, with the equivalent conversion rule:

\*If the nodelist contains one or more nodes, the conversion result is LogicalTrue.

\*If the nodelist is empty, the conversion result is LogicalFalse.

Notes:

\*Extraction of a value from a nodelist can be performed in several ways, so an implicit conversion from NodesType to ValueType may be surprising and has therefore not been defined.

\*A function expression with a declared type of NodesType can indirectly be used as an argument for a parameter of declared type ValueType by wrapping the expression in a call to a function extension, such as value() (see [Section 2.4.8](#)), that takes a parameter of type NodesType and returns a result of type ValueType.

The well-typedness of function expressions can now be defined in terms of this type system.

### 2.4.3. Well-Typedness of Function Expressions

For a function expression to be well typed:

1. its declared type must be well typed in the context in which it occurs, and
2. its arguments must be well typed for the declared type of the corresponding parameters.

(1) As per the grammar, a function expression can occur in three different immediate contexts, which lead to the following conditions for well-typedness:

**As a test-expr in a logical expression:**

The function's declared result type is LogicalType, or (giving rise to conversion as per [Section 2.4.2](#)) NodeType.

**As a comparable in a comparison:**

The function's declared result type is ValueType.

**As a function-argument in another function expression:**

The function's declared result type fulfills the following rules for the corresponding parameter of the enclosing function.

(2) The arguments of the function expression are well typed when each argument of the function can be used for the declared type of the corresponding parameter, according to one of the following conditions:

\*When the argument is a function expression with declared result type the same as the declared type of the parameter.

\*When the declared type of the parameter is LogicalType and the argument is one of the following:

- A function expression with declared result type NodeType. In this case the argument is converted to LogicalType as per [Section 2.4.2](#).

- A logical-expr that is not a function expression.

\*When the declared type of the parameter is NodeType and the argument is a query (which includes singular query).

\*When the declared type of the parameter is ValueType and the argument is one of the following:

- A value expressed as a literal.

-A singular query. In this case:

- oIf the query results in a nodelist consisting of a single node, the argument is the value of the node.

- oIf the query results in an empty nodelist, the argument is Nothing.

#### 2.4.4. `length()` Function Extension

**Parameters:** `ValueType`

**Result:** `ValueType` (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:

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

Its only argument is an instance of `ValueType` (possibly taken from a singular query, as in the example above). The result also is an instance of `ValueType`: an unsigned integer or Nothing.

- \*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.

#### 2.4.5. `count()` Function Extension

**Parameters:** `NodesType`

**Result:** `ValueType` (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:

```
$[?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: there is no deduplication of the nodelist.

#### 2.4.6. match() Function Extension

**Parameters:** ValueType (string)

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

**Result:** LogicalType

The match() function extension provides a way to check whether (the entirety of, see [Section 2.4.7](#) below) a given string matches a given regular expression, which is in [\[I-D.draft-ietf-jsonpath-iregexp\]](#) form.

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

Its arguments are instances of ValueType (possibly taken from a singular query, as for the first argument in the example above). If the first argument is not a string or the second argument is not a string conforming to [\[I-D.draft-ietf-jsonpath-iregexp\]](#), the result is LogicalFalse. Otherwise, the string that is the first argument is matched against the iregexp contained in the string that is the second argument; the result is LogicalTrue if the string matches the iregexp and LogicalFalse otherwise.

#### 2.4.7. search() Function Extension

**Parameters:** ValueType (string)

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

**Result:** LogicalType

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 arguments are instances of ValueType (possibly taken from a singular query, as for the first argument in the example above). If the first argument is not a string or the second argument is not a string conforming to [\[I-D.draft-ietf-jsonpath-iregexp\]](#), the result is LogicalFalse. Otherwise, the string that is the first argument is searched for at least one substring that matches the iregexp contained in the string that is the second argument; the result is LogicalTrue if such a substring exists and LogicalFalse otherwise.

#### 2.4.8. value() Function Extension

**Parameters:**

1. NodeType

**Result:** ValueType

The value() function extension provides a way to convert an instance of NodeType to a value and make that available for further processing in the filter expression:

```
$[?value(@..color) == "red"]
```

Its only argument is an instance of NodeType (possibly taken from a filter-query, as in the example above). The result is an instance of ValueType.

\*If the argument contains a single node, the result is the value of the node.

\*If the argument is Nothing or contains multiple nodes, the result is Nothing.

Note: a singular query may be used anywhere where a ValueType is expected, so there is no need to use the value() function extension with a singular query.

**2.4.9. Examples**

Query	Comment
<code>\$[?length(@) &lt; 3]</code>	well typed
<code>\$[?length(@.*) &lt; 3]</code>	not well typed since @.* is a non-singular query
<code>\$[?count(@.*) == 1]</code>	well typed
<code>\$[?count(1) == 1]</code>	not well typed since 1 is not a query or function expression
<code>\$[?count(foo(@.*) == 1]</code>	well typed, where foo() is a function extension with a parameter of type NodeType and result type NodeType
<code>\$[?match(@.timezone, 'Europe/.*')]</code>	well typed
<code>\$[?match(@.timezone, 'Europe/.*') == true]</code>	not well typed as LogicalType may not be used in comparisons
<code>\$[?value(@..color) == "red"]</code>	well typed
<code>\$[?value(@..color)]</code>	not well typed as ValueType may not be used in a test expression
<code>\$[?bar(@.a)]</code>	well typed for any function bar() with a parameter of any declared type and result type LogicalType

Query	Comment
<code>\$[?bml(@.*)]</code>	well typed for any function <code>bml()</code> with a parameter of declared type <code>NodeType</code> or <code>LogicalType</code> and result type <code>LogicalType</code>
<code>\$[?blt(1==1)]</code>	well typed, where <code>blt()</code> is a function with a parameter of declared type <code>LogicalType</code> and result type <code>LogicalType</code>
<code>\$[?blt(1)]</code>	not well typed for the same function <code>blt()</code> , as 1 is not a query, logical-expr, or function expression
<code>\$[?bal(1)]</code>	well typed, where <code>bal()</code> is a function with a parameter of declared type <code>ValueType</code> and result type <code>LogicalType</code>

Table 14: Function expression examples

## 2.5. Segments

For each node in an input nodelist, segments apply one or more selectors to the node and concatenate the results of each selector into per-input-node nodelists, which are then concatenated in the order of the input nodelist to form a single segment result 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  $N \geq 0$ , produces a nodelist consisting of nodes at depth in the input value of N or greater.

\*A query with N segments, where  $N \geq 0$ , all of which are [child segments](#) ([Section 2.5.1](#)), 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.

### 2.5.1. 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: 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.

For each node in the input nodelist, 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: 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:

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

Table 15: Child segment examples

### 2.5.2. 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>'`. Note: as with the similar shorthand of a child-segment, this can only be used with member names that are composed of certain characters, as specified in the ABNF rule member-name-shorthand.

Note: `..` on its own is not a valid segment.

#### Semantics

A descendant segment produces zero or more descendants of an input value.

For each node in the input nodelist, a descendant selector visits the input node 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  $D_1, \dots, D_n$  (where  $n \geq 1$ ). Note:  $D_1$  is the input value.

For each  $i$  such that  $1 \leq i \leq n$ , the nodelist  $R_i$  is defined to be a result of applying the child segment `[<selectors>]` to the node  $D_i$ .

For each node in the input nodelist, the result of the descendant segment is the concatenation of  $R_1, \dots, R_n$  (in that order). These results are then concatenated in input nodelist order to form the result of the segment.

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

## Examples

JSON:

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

Queries:

Query	Result	Result Paths	Comment
<code>\$.j</code>	1 4	<code>\$['o']['j']</code> <code>\$['a'][2][0]</code> <code>['j']</code>	Object values
<code>\$.j</code>	4 1	<code>\$['a'][2][0]</code> <code>['j']</code> <code>\$['o']['j']</code>	Alternative result
<code>\$.[0]</code>	5 { "j": 4 }	<code>\$['a'][0]</code> <code>\$['a'][2][0]</code>	Array values
<code>\$.[*]</code> <code>\$.*</code>	{ "j": 1, "k" : 2 } [5, 3, [{ "j": 4 }, { "k": 6 }]] 1 2 5 3 [{"j": 4}, {"k":	<code>\$['o']</code> <code>\$['a']</code> <code>\$['o']['j']</code> <code>\$['o']['k']</code> <code>\$['a'][0]</code> <code>\$['a'][1]</code> <code>\$['a'][2]</code> <code>\$['a'][2][0]</code>	All values

Query	Result	Result Paths	Comment
	6}} {"j": 4} {"k": 6} 4 6	\$['a'][2][1] \$['a'][2][0] ['j'] \$['a'][2][1] ['k']	
\$.o	{"j": 1, "k": 2}	\$['o']	Input value is visited
\$.o..[*, *]	1 2 2 1	\$['o']['j'] \$['o']['k'] \$['o']['k'] \$['o']['j']	Non-deterministic ordering
\$.a..[0, 1]	5 3 {"j": 4} {"k": 6}	\$['a'][0] \$['a'][1] \$['a'][2][0] \$['a'][2][1]	Multiple segments

Table 16: Descendant segment 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}, {"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 {"k": 6},
- \*{"k": 6} must appear before 4, and
- \*4 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).

## 2.6. Semantics of null

Note: 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 member name string

Table 17: Examples involving (or not involving) null

## 2.7. Normalized Paths

A Normalized Path is a unique representation of the location of a node in a value which 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: 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 nodelist may be represented compactly in JSON as an array 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 in Normalized Paths 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 Normalized Paths, in one and only one way; all other characters are unescaped.

Note: Normalized Paths are singular queries, but not all singular queries are Normalized Paths. For example, `$[-3]` is a singular query, but is not a Normalized Path. The Normalized Path equivalent to `$[-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.)

```

normalized-path      = root-identifier *(normal-index-segment)
normal-index-segment = "[" normal-selector "]"
normal-selector      = normal-name-selector / normal-index-selector
normal-name-selector = %x27 *normal-single-quoted %x27 ; 'string'
normal-single-quoted = normal-unescaped /
                        ESC normal-escapable
normal-unescaped     =          ; omit %x0-1F control codes
                        %x20-26 /
                                 ; omit 0x27 '
                        %x28-5B /
                                 ; omit 0x5C \
                        %x5D-10FFFF
normal-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
                        "'" / ; ' 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"
                            ; omit U+0008-U+000A BS HT LF
                          ("0" %x62) /      ; "0b"
                            ; omit U+000C-U+000D FF CR
                          ("0" %x65-66) / ; "0e"-"0f"
                          ("1" normal-HEXDIG)
                        )
normal-HEXDIG        = DIGIT / %x61-66      ; "0"-"9", "a"-"f"
normal-index-selector = "0" / (DIGIT1 *DIGIT)
                        ; non-negative decimal integer

```

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

Path	Normalized Path	Comment
<code>\$.a</code>	<code>\$['a']</code>	Object value
<code>\$.1</code>	<code>\$.1</code>	Array index
<code>\$. -3</code>	<code>\$.2</code>	

Path	Normalized Path	Comment
		Negative array index for an array of length 5
<code>\$.a.b[1:2]</code>	<code>\$['a']['b'][1]</code>	Nested structure
<code>\$["\u000B"]</code>	<code>\$['\u000b']</code>	Unicode escape
<code>\$["\u0061"]</code>	<code>\$['a']</code>	Unicode character

Table 18: Normalized Path examples

### 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](#) of [\[BCP26\]](#)).

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](#) of [\[BCP26\]](#)), 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 sub-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]*)`. No other entry in the sub-registry can have the same function name.

**Brief description:**

a brief description

**Parameters:**

A comma-separated list of zero or more declared types, one for each of the arguments expected for this function extension

**Result:**

The declared type of the result for this function extension

**Change Controller:**

(see [Section 2.3](#) of [[BCP26](#)])

**Reference:**

a reference document that provides a description of the function extension

Initial entries in this sub-registry are as listed in [Table 19](#); the Column "Change Controller" always has the value "IESG" and the column "Reference" always has the value "[Section 2.4](#) of RFCthis":

Function Name	Brief description	Parameters	Result
length	length of string, array, object	ValueType	ValueType
count	size of nodelist	NodesType	ValueType
match	regular expression full match	ValueType, ValueType	LogicalType
search	regular expression substring match	ValueType, ValueType	LogicalType
value	value of single node in nodelist	NodesType	ValueType

Table 19: 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 query 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 a query argument with certain constructs described as unpredictable by [\[RFC8259\]](#).

Predictable behavior can be expected, except in relation to the ordering of objects, for any query 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

- [BCP26] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/rfc/rfc8126>>.
- [I-D.draft-ietf-jsonpath-iregexp] Bormann, C. and T. Bray, "I-Regexp: An Interoperable Regexp Format", Work in Progress, Internet-Draft, draft-ietf-jsonpath-iregexp-04, 31 March 2023, <<https://datatracker.ietf.org/doc/html/draft-ietf-jsonpath-iregexp-04>>.
- [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/rfc/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/rfc/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/rfc/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/rfc/rfc6838>>.
- [RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <<https://www.rfc-editor.org/rfc/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/rfc/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/rfc/rfc8259>>.

**[STD80]**

Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <<https://www.rfc-editor.org/rfc/rfc20>>.

**[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>>.

## 5.2. Informative References

**[BOOLEAN-LAWS]** "Boolean algebra laws", n.d., <[https://en.wikipedia.org/wiki/Boolean\\_algebra#Laws](https://en.wikipedia.org/wiki/Boolean_algebra#Laws)>.

**[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/rfc/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/rfc/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: 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 query 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 query 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 ?<logical-expr> as in

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

[Table 20](#) 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

XPath	JSONPath	Description
		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 20: XPath syntax compared to JSONPath

For further illustration, [Table 21](#) 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
//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 21: 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 *odelist* resulting from the previous query segment. 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 reference token (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](mailto:mmikulicic@gmail.com)

Edward Surov  
TheSoul Publishing Ltd.  
Limassol  
Cyprus

Email: [esurov.tsp@gmail.com](mailto:esurov.tsp@gmail.com)

Greg Dennis  
Auckland



New Zealand

Email: [gregsdennis@yahoo.com](mailto: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](mailto:stefan.goessner@fh-dortmund.de)

Glyn Normington (editor)  
Winchester  
United Kingdom

Email: [glyn.normington@gmail.com](mailto:glyn.normington@gmail.com)

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

Phone: [+49-421-218-63921](tel:+49-421-218-63921)

Email: [cabo@tzi.org](mailto:cabo@tzi.org)