Package 'rjsoncons'

March 15, 2025

Title Query, Pivot, Patch, and Validate 'JSON' and 'NDJSON'

Version 1.3.2

Description Functions to query (filter or transform), pivot (convert from array-of-objects to object-of-arrays, for easy import as 'R' data frame), search, patch (edit), and validate (against 'JSON Schema') 'JSON' and 'NDJSON' strings, files, or URLs. Query and pivot support 'JSONpointer', 'JSONpath' or 'JMESpath' expressions. The implementation uses the 'jsoncons' <https://danielaparker.github.io/jsoncons/> header-only library; the library is easily linked to other packages for direct access to 'C++' functionality not implemented here.

Depends R (>= 4.2.0)

Imports cli, tibble

Suggests jsonlite, tinytest, BiocStyle, knitr, rmarkdown

LinkingTo cpp11, cli

License BSL-1.0

NeedsCompilation yes

Encoding UTF-8

BugReports https://github.com/mtmorgan/rjsoncons/issues

RoxygenNote 7.3.1

VignetteBuilder knitr

URL https://mtmorgan.github.io/rjsoncons/

Author Martin Morgan [aut, cre] (<https://orcid.org/0000-0002-5874-8148>), Marcel Ramos [aut] (<https://orcid.org/0000-0002-3242-0582>), Daniel Parker [aut, cph] (jsoncons C++ library maintainer)

Maintainer Martin Morgan <mtmorgan.xyz@gmail.com>

Repository CRAN

Date/Publication 2025-03-15 17:50:03 UTC

Contents

as_r 2
jsonpath 3
j_data_type 6
j_flatten 7
j_patch_apply 11
j_query 14
j_schema_is_valid 16
version 18

Index **19**

`as_r` *Parse JSON or NDJSON to R*

Description

`as_r()` transforms JSON or NDJSON to an *R* object.

Usage

```
as_r(
  data,
  object_names = "asis",
  ...,
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data)
)
```

Arguments

`data`	a character() JSON string or NDJSON records, or the name of a file or URL containing JSON or NDJSON, or an *R* object parsed to a JSON string using `jsonlite::toJSON()`.
`object_names`	character(1) order `data` object elements `"asis"` (default) or `"sort"` before filtering on `path`.
`...`	passed to `jsonlite::toJSON` when `data` is an *R* object.
`n_records`	numeric(1) maximum number of NDJSON records parsed.
`verbose`	logical(1) report progress when parsing large NDJSON files.
`data_type`	character(1) type of `data`; one of `"json"`, `"ndjson"`, or a value returned by `j_data_type()`.

Details

The `as = "R"` argument to `j_query()`, `j_pivot()`, and the `as_r()` function transform JSON or NDJSON to an *R* object. JSON and NDJSON can be a character vector, file, or url, or an *R* object (which is first translated to a JSON string). Main rules are:

- JSON arrays of a single type (boolean, integer, double, string) are transformed to *R* vectors of the same length and corresponding type. A JSON scalar and a JSON vector of length 1 are represented in the same way in *R*.
- If a JSON 64-bit integer array contains a value larger than *R*'s 32-bit integer representation, the array is transformed to an *R* numeric vector. NOTE that this results in loss of precision for 64-bit integer values greater than `2^53`.
- JSON arrays mixing integer and double values are transformed to *R* numeric vectors.
- JSON objects are transformed to *R* named lists.

The vignette reiterates this information and provides additional details.

Value

`as_r()` returns an *R* object.

Examples

```
## as_r()
as_r('[1, 2, 3]')       # JSON integer array -> R integer vector
as_r('[1, 2.0, 3]')     # JSON intger and double array -> R numeric vector
as_r('[1, 2.0, "3"]')   # JSON mixed array -> R list
as_r('[1, 2147483648]') # JSON integer > R integer max -> R numeric vector

json <- '{"b": 1, "a": ["c", "d"], "e": true, "f": [true], "g": {}}'
as_r(json) |> str()     # parsing complex objects
identical(              # JSON scalar and length 1 array identical in R
    as_r('{"a": 1}'), as_r('{"a": [1]}')
)
```

`jsonpath` *JSONpath, JMESpath, or JSONpointer query of JSON / NDJSON documents; use* `j_query()` *instead*

Description

`jsonpath()` executes a query against a JSON string or vector NDJSON entries using the 'JSONpath' specification.

`jmespath()` executes a query against a JSON string using the 'JMESpath' specification.

`jsonpointer()` extracts an element from a JSON string using the 'JSON pointer' specification.

Usage

```
jsonpath(data, path, object_names = "asis", as = "string", ...)

jmespath(data, path, object_names = "asis", as = "string", ...)

jsonpointer(data, path, object_names = "asis", as = "string", ...)
```

Arguments

`data`	a character() JSON string or NDJSON records, or the name of a file or URL containing JSON or NDJSON, or an *R* object parsed to a JSON string using `jsonlite::toJSON()`.
`path`	character(1) JSONpointer, JSONpath or JMESpath query string.
`object_names`	character(1) order `data` object elements `"asis"` (default) or `"sort"` before filtering on `path`.
`as`	character(1) return type. `"string"` returns a single JSON string; `"R"` returns an *R* object following the rules outlined for `as_r()`.
`...`	arguments for parsing NDJSON, or passed to `jsonlite::toJSON` when `data` is not character-valued. For NDJSON, • Use `n_records = 2` to parse just the first two records of the NDJSON document. • Use `verbose = TRUE` to obtain a progress bar when reading from a connection (file or URL). Requires the cli package. As an example for use with `jsonlite::toJSON()` • use `auto_unbox = TRUE` to automatically 'unbox' vectors of length 1 to JSON scalar values.

Value

`jsonpath()`, `jmespath()` and `jsonpointer()` return a character(1) JSON string (`as = "string"`, default) or *R* object (`as = "R"`) representing the result of the query.

Examples

```
json <- '{
  "locations": [
    {"name": "Seattle", "state": "WA"},
    {"name": "New York", "state": "NY"},
    {"name": "Bellevue", "state": "WA"},
    {"name": "Olympia", "state": "WA"}
  ]
 }'

## return a JSON string
jsonpath(json, "$..name") |>
    cat("\n")
```

```
## return an R object
jsonpath(json, "$..name", as = "R")

## create a list with state and name as scalar vectors
lst <- as_r(json)

if (requireNamespace("jsonlite", quietly = TRUE)) {
## objects other than scalar character vectors are automatically
## coerced to JSON; use `auto_unbox = TRUE` to represent R scalar
## vectors in the object as JSON scalar vectors
jsonpath(lst, "$..name", auto_unbox = TRUE) |>
    cat("\n")

## use I("Seattle") to coerce to a JSON object ["Seattle"]
jsonpath(I("Seattle"), "$[0]")      |> cat("\n")
}

## a scalar character vector like "Seattle" is not valid JSON...
try(jsonpath("Seattle", "$"))
## ...but a double-quoted string is
jsonpath('"Seattle"', "$")

## different ordering of object names -- 'asis' (default) or 'sort'
json_obj <- '{"b": "1", "a": "2"}'
jsonpath(json_obj, "$")           |> cat("\n")
jsonpath(json_obj, "$.*")         |> cat("\n")
jsonpath(json_obj, "$", "sort")   |> cat("\n")
jsonpath(json_obj, "$.*", "sort") |> cat("\n")

path <- "locations[?state == 'WA'].name | sort(@)"
jmespath(json, path) |>
    cat("\n")

if (requireNamespace("jsonlite", quietly = TRUE)) {
## original filter always fails, e.g., '["WA"] != 'WA'
jmespath(lst, path)  # empty result set, '[]'

## filter with unboxed state, and return unboxed name
jmespath(lst, "locations[?state[0] == 'WA'].name[0] | sort(@)") |>
    cat("\n")

## automatically unbox scalar values when creating the JSON string
jmespath(lst, path, auto_unbox = TRUE) |>
    cat("\n")
}

## jsonpointer 0-based arrays
jsonpointer(json, "/locations/0/name")

## document root "", sort selected element keys
jsonpointer('{"b": 0, "a": 1}', "", "sort", as = "R") |>
    str()
```

```
## 'Key not found' -- path '/' searches for a 0-length key
try(jsonpointer('{"b": 0, "a": 1}', "/"))
```

`j_data_type` *Detect JSON and NDJSON data and path types*

Description

`j_data_type()` uses simple rules to determine whether 'data' is JSON, NDJSON, file, url, or R.

`j_path_type()` uses simple rules to identify whether `path` is a JSONpointer, JSONpath, or JMESpath expression.

Usage

```
j_data_type(data)

j_path_type(path)
```

Arguments

`data`	a character() JSON string or NDJSON records, or the name of a file or URL containing JSON or NDJSON, or an *R* object parsed to a JSON string using `jsonlite::toJSON()`.
`path`	character(1) JSONpointer, JSONpath or JMESpath query string.

Details

`j_data_type()` without any arguments reports possible return values: `"json"`, `"ndjson"`, `"file"`, `"url"`, `"R"`. When provided an argument, `j_data_type()` infers (but does not validate) the type of `data` based on the following rules:

- For a scalar (length 1) character `data`, either `"url"` (matching regular expression `"^https?://"`, `"file"` (`file.exists(data)` returns `TRUE`), or `"json"`. When `"file"` or `"url"` is inferred, the return value is a length 2 vector, with the first element the inferred type of data (`"json"` or `"ndjson"`) obtained from the first 2 lines of the file.
- For character data with `length(data) > 1`, `"ndjson"` if all elements start a square bracket or curly brace, consistently (i.e., agreeing with the start of the first record), otherwise `"json"`.
- `"R"` for all non-character data.

`j_path_type()` without any argument reports possible values: `"JSONpointer"`, `"JSONpath"`, or `"JMESpath"`. When provided an argument, `j_path_type()` infers the type of `path` using a simple but incomplete classification:

- `"JSONpointer"` is inferred if the the path is `""` or starts with `"/"`.
- `"JSONpath"` expressions start with `"$"`.
- `"JMESpath"` expressions satisfy neither the `JSONpointer` nor `JSONpath` criteria.

Because of these rules, the valid JSONpointer path `"@"` is interpreted as JMESpath; use `jsonpointer()` if JSONpointer behavior is required.

Examples

```
j_data_type()                           # available types
j_data_type("")                         # json
j_data_type('{"a": 1}')                 # json
j_data_type(c('[{"a": 1}', '{"a": 2}]')) # json
j_data_type(c('{"a": 1}', '{"a": 2}'))   # ndjson
j_data_type(list(a = 1, b = 2))         # R
fl <- system.file(package = "rjsoncons", "extdata", "example.json")
j_data_type(fl)                         # c('json', 'file')
j_data_type(readLines(fl))              # json

j_path_type()                           # available types
j_path_type("")                         # JSONpointer
j_path_type("/locations/0/name")        # JSONpointer
j_path_type("$.locations[0].name")      # JSONpath
j_path_type("locations[0].name")        # JMESpath
```

`j_flatten` *Flatten and find keys or values in JSON or NDJSON documents*

Description

`j_flatten()` transforms a JSON document into a list where names are JSONpointer 'paths' and elements are the corresponding 'values' from the JSON document.

`j_find_values()` finds paths to exactly matching values.

`j_find_values_grep()` finds paths to values matching a regular expression.

`j_find_keys()` finds paths to exactly matching keys.

`j_find_keys_grep()` finds paths to keys matching a regular expression.

For NDJSON documents, the result is either a character vector (for `as = "string"`) or list of *R* objects, one element for each NDJSON record.

Usage

```
j_flatten(
  data,
  object_names = "asis",
  as = "string",
  ...,
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = "JSONpointer"
)

j_find_values(
```

```
  data,
  values,
  object_names = "asis",
  as = "R",
  ...,
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = "JSONpointer"
)

j_find_values_grep(
  data,
  pattern,
  object_names = "asis",
  as = "R",
  ...,
  grep_args = list(),
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = "JSONpointer"
)

j_find_keys(
  data,
  keys,
  object_names = "asis",
  as = "R",
  ...,
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = "JSONpointer"
)

j_find_keys_grep(
  data,
  pattern,
  object_names = "asis",
  as = "R",
  ...,
  grep_args = list(),
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = "JSONpointer"
)
```

Arguments

data	a character() JSON string or NDJSON records, or the name of a file or URL containing JSON or NDJSON, or an *R* object parsed to a JSON string using jsonlite::toJSON().
object_names	character(1) order data object elements "asis" (default) or "sort" before filtering on path.
as	character(1) describing the return type. For j_flatten(), either "string" or "R". For other functions on this page, one of "R", "data.frame", or "tibble".
...	passed to jsonlite::toJSON when data is an *R* object.
n_records	numeric(1) maximum number of NDJSON records parsed.
verbose	logical(1) report progress when parsing large NDJSON files.
data_type	character(1) type of data; one of "json", "ndjson", or a value returned by j_data_type().
path_type	character(1) type of 'path' to be returned; one of '"JSONpointer"', '"JSONpath"'; '"JMESpath"' is not supported.
values	vector of one or more values to be matched exactly to values in the JSON document.
pattern	character(1) regular expression to match values or paths.
grep_args	list() additional arguments passed to grepl() when searching on values or paths.
keys	character() vector of one or more keys to be matched exactly to path elements.

Details

Functions documented on this page expand data into all path / value pairs. This is not suitable for very large JSON documents.

For j_find_keys(), the key must exactly match one or more consecutive keys in the JSONpointer path returned by j_flatten().

For j_find_keys_grep(), the key can define a pattern that spans across JSONpointer or JSONpath elements.

Value

j_flatten(as = "string") (default) returns a JSON string representation of the flattened document, i.e., an object with keys the JSONpointer paths and values the value at the corresponding path in the original document.

j_flatten(as = "R") returns a named list, where names() are the JSONpointer paths to each element in the JSON document and list elements are the corresponding values.

j_find_values() and j_find_values_grep() return a list with names as JSONpointer paths and list elements the matching values, or a data.frame or tibble with columns path and value. Values are coerced to a common type when as is data.frame or tibble.

j_find_keys() and j_find_keys_grep() returns a list, data.frame, or tibble similar to j_find_values() and j_find_values_grep().

For NDJSON documents, the result is a vector paralleling the NDJSON document, with j_flatten() applied to each element of the NDJSON document.

Examples

```
json <- '{
    "discards": {
        "1000": "Record does not exist",
        "1004": "Queue limit exceeded",
        "1010": "Discarding timed-out partial msg"
    },
    "warnings": {
        "0": "Phone number missing country code",
        "1": "State code missing",
        "2": "Zip code missing"
    }
}'

## JSONpointer
j_flatten(json) |>
    cat("\n")

## JSONpath
j_flatten(json, as = "R", path_type = "JSONpath") |>
    str()

j_find_values(json, "Zip code missing", as = "tibble")
j_find_values(
    json,
    c("Queue limit exceeded", "Zip code missing"),
    as = "tibble"
)

j_find_values_grep(json, "missing", as = "tibble")

## JSONpath
j_find_values_grep(json, "missing", as = "tibble", path_type = "JSONpath")

j_find_keys(json, "discards", as = "tibble")
j_find_keys(json, "1", as = "tibble")
j_find_keys(json, c("discards", "warnings"), as = "tibble")

## JSONpath
j_find_keys(json, "discards", as = "tibble", path_type = "JSONpath")

j_find_keys_grep(json, "discard", as = "tibble")
j_find_keys_grep(json, "1", as = "tibble")
j_find_keys_grep(json, "car.*/101", as = "tibble")

## JSONpath
j_find_keys_grep(json, "car.*\\['101", as = "tibble", path_type = "JSONpath")

## NDJSON

ndjson_file <-
    system.file(package = "rjsoncons", "extdata", "example.ndjson")
```

```
j_flatten(ndjson_file) |>
    noquote()
j_find_values_grep(ndjson_file, "e") |>
    str()
```

`j_patch_apply` *Patch or compute the difference between two JSON documents*

Description

`j_patch_apply()` uses JSON Patch https://jsonpatch.com to transform JSON 'data' according the rules in JSON 'patch'.

`j_patch_from()` computes a JSON patch describing the difference between two JSON documents.

`j_patch_op()` translates *R* arguments to the JSON representation of a patch, validating and 'unboxing' arguments as necessary.

Usage

```
j_patch_apply(data, patch, as = "string", ...)

j_patch_from(data_x, data_y, as = "string", ...)

j_patch_op(op, path, ...)

## Default S3 method:
j_patch_op(op, path, ..., from = NULL, value = NULL)

## S3 method for class 'j_patch_op'
j_patch_op(op, ...)

## S3 method for class 'j_patch_op'
c(..., recursive = FALSE)

## S3 method for class 'j_patch_op'
print(x, ...)
```

Arguments

`data`	JSON character vector, file, URL, or an *R* object to be converted to JSON using `jsonline::fromJSON(data, ...)`.
`patch`	JSON 'patch' as character vector, file, URL, *R* object, or the result of `j_patch_op()`.
`as`	character(1) return type; `"string"` returns a JSON string, `"R"` returns an *R* object using the rules in `as_r()`.

...	For j_patch_apply() and j_patch_diff(), arguments passed to jsonlite::toJSON when data, patch, data_x, and / or data_y is an *R* object. It is appropriate to add the jsonlite::toJSON() argument auto_unbox = TRUE when patch is an *R* object and any 'value' fields are JSON scalars; for more complicated scenarios 'value' fields should be marked with jsonlite::unbox() before being passed to j_patch_*(). For j_patch_op() the ... are additional arguments to the patch operation, e.g., path = ', value = '.
data_x	As for data.
data_y	As for data.
op	A patch operation ("add", "remove", "replace", "copy", "move", "test"), or when 'piping' an object created by j_patch_op().
path	A character(1) JSONPointer path to the location being patched.
from	A character(1) JSONPointer path to the location an object will be copied or moved from.
value	An *R* object to be translated into JSON and used during add, replace, or test.
recursive	Ignored.
x	An object produced by j_patch_op().

Details

For j_patch_apply(), 'patch' is a JSON array of objects. Each object describes how the patch is to be applied. Simple examples are available at https://jsonpatch.com, with verbs 'add', 'remove', 'replace', 'copy' and 'test'. The 'path' element of each operation is a JSON pointer; remember that JSON arrays are 0-based.

- add – add elements to an existing document.

```
{"op": "add", "path": "/biscuits/1", "value": {"name": "Ginger Nut"}}
```

- remove – remove elements from a document.

```
{"op": "remove", "path": "/biscuits/0"}
```

- replace – replace one element with another

```
{
    "op": "replace", "path": "/biscuits/0/name",
    "value": "Chocolate Digestive"
}
```

- copy – copy a path to another location.

```
{"op": "copy", "path": "/best_biscuit", "from": "/biscuits/0"}
```

- move – move a path to another location.

```
{"op": "move", "path": "/cookies", "from": "/biscuits"}
```

- test – test for the existence of a path; if the path does not exist, do not apply any of the patch.

```
{"op": "test", "path": "/best_biscuit/name", "value": "Choco Leibniz"}
```

The examples below illustrate a patch with one (a JSON array with a single object) or several (a JSON array with several arguments) operations. `j_patch_apply()` fits naturally into a pipeline composed with `|>` to transform JSON between representations.

The `j_patch_op()` function takes care to ensure that `op`, `path`, and `from` arguments are 'unboxed' (represented as JSON scalars rather than arrays). The user must ensure that `value` is represented correctly by applying `jsonlite::unbox()` to individual elements or adding `auto_unbox = TRUE` to `...`. Examples illustrate these different scenarios.

Value

`j_patch_apply()` returns a JSON string or *R* object representing 'data' patched according to 'patch'.

`j_patch_from()` returns a JSON string or *R* object representing the difference between 'data_x' and 'data_y'.

`j_patch_op()` returns a character vector subclass that can be used in `j_patch_apply()`.

Examples

```
data_file <-
    system.file(package = "rjsoncons", "extdata", "patch_data.json")

## add a biscuit
patch <- '[
    {"op": "add", "path": "/biscuits/1", "value": {"name": "Ginger Nut"}}
]'
j_patch_apply(data_file, patch, as = "R") |> str()

## add a biscuit and choose a favorite
patch <- '[
    {"op": "add", "path": "/biscuits/1", "value": {"name": "Ginger Nut"}},
    {"op": "copy", "path": "/best_biscuit", "from": "/biscuits/2"}
]'
biscuits <- j_patch_apply(data_file, patch)
as_r(biscuits) |> str()

j_patch_from(biscuits, data_file, as = "R") |> str()

if (requireNamespace("jsonlite", quietly = TRUE)) {
## helper for constructing patch operations from R objects
j_patch_op(
    "add", path = "/biscuits/1", value = list(name = "Ginger Nut"),
    ## 'Ginger Nut' is a JSON scalar, so auto-unbox the 'value' argument
    auto_unbox = TRUE
)
j_patch_op("remove", "/biscuits/0")
j_patch_op(
    "replace", "/biscuits/0/name",
    ## also possible to unbox arguments explicitly
    value = jsonlite::unbox("Chocolate Digestive")
```

```
)
j_patch_op("copy", "/best_biscuit", from = "/biscuits/0")
j_patch_op("move", "/cookies", from = "/biscuits")
j_patch_op(
    "test", "/best_biscuit/name", value = "Choco Leibniz",
    auto_unbox = TRUE
)

## several operations
value <- list(name = jsonlite::unbox("Ginger Nut"))
ops <- c(
    j_patch_op("add", "/biscuits/1", value = value),
    j_patch_op("copy", path = "/best_biscuit", from = "/biscuits/0")
)
ops

ops <-
    j_patch_op("add", "/biscuits/1", value = value) |>
    j_patch_op("copy", path = "/best_biscuit", from = "/biscuits/0")
ops
}
```

`j_query` *Query and pivot JSON and NDJSON documents*

Description

`j_query()` executes a query against a JSON or NDJSON document, automatically inferring the type of `data` and `path`.

`j_pivot()` transforms a JSON array-of-objects to an object-of-arrays; this can be useful when forming a column-based tibble from row-oriented JSON / NDJSON.

Usage

```
j_query(
  data,
  path = "",
  object_names = "asis",
  as = "string",
  ...,
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = j_path_type(path)
)

j_pivot(
  data,
```

```
  path = "",
  object_names = "asis",
  as = "string",
  ...,
  n_records = Inf,
  verbose = FALSE,
  data_type = j_data_type(data),
  path_type = j_path_type(path)
)
```

Arguments

data	a character() JSON string or NDJSON records, or the name of a file or URL containing JSON or NDJSON, or an *R* object parsed to a JSON string using `jsonlite::toJSON()`.
path	character(1) JSONpointer, JSONpath or JMESpath query string.
object_names	character(1) order `data` object elements `"asis"` (default) or `"sort"` before filtering on `path`.
as	character(1) return type. For `j_query()`, `"string"` returns JSON / NDJSON strings; `"R"` parses JSON / NDJSON to R using rules in `as_r()`. For `j_pivot()` (JSON only), use `as = "data.frame"` or `as = "tibble"` to coerce the result to a data.frame or tibble.
...	passed to `jsonlite::toJSON` when `data` is an *R* object.
n_records	numeric(1) maximum number of NDJSON records parsed.
verbose	logical(1) report progress when parsing large NDJSON files.
data_type	character(1) type of `data`; one of `"json"`, `"ndjson"`, or a value returned by `j_data_type()`.
path_type	character(1) type of `path`; one of `"JSONpointer"`, `"JSONpath"`, `"JMESpath"`. Inferred from `path` using `j_path_type()`.

Details

`j_pivot()` transforms an 'array-of-objects' (typical when the JSON is a row-oriented representation of a table) to an 'object-of-arrays'. A simple example transforms an array of two objects each with three fields `'[{"a": 1, "b": 2, "c": 3}, {"a": 4, "b": 5, "c": 6}]'` to an object with three fields, each a vector of length 2 `'{"a": [1, 4], "b": [2, 5], "c": [3, 6]}'`. The object-of-arrays representation corresponds closely to an *R* data.frame or tibble, as illustrated in the examples.

`j_pivot()` with JMESpath paths are especially useful for transforming NDJSON to a `data.frame` or `tibble`

Examples

```
json <- '{
  "locations": [
    {"name": "Seattle", "state": "WA"},
    {"name": "New York", "state": "NY"},
    {"name": "Bellevue", "state": "WA"},
```

```
    {"name": "Olympia", "state": "WA"}
  ]
}'

j_query(json, "/locations/0/name")             # JSONpointer
j_query(json, "$.locations[*].name", as = "R") # JSONpath
j_query(json, "locations[].state", as = "R")   # JMESpath

## a few NDJSON records from <https://www.gharchive.org/>
ndjson_file <-
    system.file(package = "rjsoncons", "extdata", "2023-02-08-0.json")
j_query(ndjson_file, "{id: id, type: type}")

j_pivot(json, "$.locations[?@.state=='WA']", as = "string")
j_pivot(json, "locations[?@.state=='WA']", as = "R")
j_pivot(json, "locations[?@.state=='WA']", as = "data.frame")
j_pivot(json, "locations[?@.state=='WA']", as = "tibble")

## use 'path' to pivot ndjson one record at at time
j_pivot(ndjson_file, "{id: id, type: type}", as = "data.frame")

## 'org' is a nested element; extract it
j_pivot(ndjson_file, "org", as = "data.frame")

## use j_pivot() to filter 'PushEvent' for organizations
path <- "[{id: id, type: type, org: org}]
             [?@.type == 'PushEvent' && @.org != null] |
                 [0]"
j_pivot(ndjson_file, path, as = "data.frame")

## try also
##
##     j_pivot(ndjson_file, path, as = "tibble") |>
##         tidyr::unnest_wider("org", names_sep = ".")
```

`j_schema_is_valid` *Validate JSON documents against JSON Schema*

Description

`j_schema_is_vaild()` uses JSON Schema https://json-schema.org/ to validate JSON 'data' according to 'schema'.

`j_schema_validate()` returns a JSON or *R* object, data.frame, or tibble, describing how `data` does not conform to `schema`. See the "Using 'jsoncons' in R" vignette for help interpreting validation results.

Usage

```
j_schema_is_valid(
```

```
  data,
  schema,
  ...,
  data_type = j_data_type(data),
  schema_type = j_data_type(schema)
)

j_schema_validate(
  data,
  schema,
  as = "string",
  ...,
  data_type = j_data_type(data),
  schema_type = j_data_type(schema)
)
```

Arguments

data	JSON character vector, file, or URL defining document to be validated. NDJSON data and schema are not supported.
schema	JSON character vector, file, or URL defining the schema against which data will be validated.
...	passed to jsonlite::toJSON when data is not character-valued.
data_type	character(1) type of data; one of "json" or a value returned by j_data_type(); schema validation does not support "ndjson" data.
schema_type	character(1) type of schema; see data_type.
as	for j_schema_validate(), one of "string", "R", "data.frame", "tibble", or "details", to determine the representation of the return value.

Examples

```
## Allowable `data_type=` and `schema_type` -- excludes 'ndjson'
j_data_type() |>
    Filter(\(type) !"ndjson" %in% type, x = _) |>
    str()
## compare JSON patch to specification. 'op' key should have value
## 'add'; 'paths' key should be key 'path'
## schema <- "https://json.schemastore.org/json-patch.json"
schema <- system.file(package = "rjsoncons", "extdata", "json-patch.json")
op <- '[{
    "op": "adds", "paths": "/biscuits/1",
    "value": { "name": "Ginger Nut" }
}]'
j_schema_is_valid(op, schema)

j_schema_validate(op, schema, as = "details")
```

version *Version of jsoncons C++ library*

Description

version() reports the version of the C++ jsoncons library in use.

Usage

```
version()
```

Value

version() returns a character(1) major.minor.patch version string, possibly with git hash for between-release version.

Examples

```
version()
```

Index

as_r, 2

c.j_patch_op *(*j_patch_apply*)*, 11

flatten_NDJSON *(*j_flatten*)*, 7

j_data_type, 6
j_find_keys *(*j_flatten*)*, 7
j_find_keys_grep *(*j_flatten*)*, 7
j_find_values *(*j_flatten*)*, 7
j_find_values_grep *(*j_flatten*)*, 7
j_flatten, 7
j_patch_apply, 11
j_patch_from *(*j_patch_apply*)*, 11
j_patch_op *(*j_patch_apply*)*, 11
j_path_type *(*j_data_type*)*, 6
j_pivot *(*j_query*)*, 14
j_query, 14
j_schema_is_valid, 16
j_schema_validate *(*j_schema_is_valid*)*, 16
jmespath *(*jsonpath*)*, 3
jsonpath, 3
jsonpointer *(*jsonpath*)*, 3

print.j_patch_op *(*j_patch_apply*)*, 11

version, 18