Package 'rjsoncons'

January 26, 2024

Title 'C++' Header-Only 'jsoncons' Library for 'JSON' Queries

Version 1.2.0

Description The 'jsoncons' <https://danielaparker.github.io/jsoncons/> 'C++' header-only library constructs representations from a 'JSON' character vector, and provides extensions for flexible queries and other operations on 'JSON' objects. This package provides 'R' functions to query (filter or transform) and pivot (convert from array-of-objects to object-of-arrays, for easy import into 'R') 'JSON' or 'NDJSON' strings or files using 'JSONpointer', 'JSONpath' or 'JMESpath' expression. The 'jsoncons' library is also be easily linked to other packages for direct access to 'C++' functionality.

Imports utils

Suggests jsonlite, tibble, cli, tinytest, BiocStyle, knitr, rmarkdown

LinkingTo cpp11

License BSL-1.0

NeedsCompilation yes

Encoding UTF-8

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

RoxygenNote 7.3.0

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 2024-01-26 22:30:02 UTC

R topics documented:

`as_r` *Parse JSON to R*

Description

`as_r()` transforms a JSON string to an *R* object.

Usage

```
as_r(data, object_names = "asis", ...)
```

Arguments

`data`	a character(1) JSON string or (unusually) an `R` object.
`object_names`	character(1) order `data` object elements `"asis"` (default) or `"sort"` before filtering on `path`.
`...`	passed to `jsonlite::toJSON()` in the unusual circumstance that `data` is an `R` object.

Details

The `as = "R"` argument to `j_query()`, `j_pivot()`, etc., and the `as_r()` function transform a JSON string representation to an *R* object. 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*

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 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 <- jsonlite::fromJSON(json, simplifyVector = FALSE)

## 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")

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

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

## 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")

## 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 / 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 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_query *Query and pivot for JSON / 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.

Usage

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

j_pivot(
  data,
  path = "",
  object_names = "asis",
  as = "string",
  ...,
  data_type = j_data_type(data),
  path_type = j_path_type(path)
)
```

Arguments

`data`	a character() JSON string or NDJSON records, 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.
`...`	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.

`data_type` character(1) type of `data`; one of `"json"`, `"ndjson"`. Inferred from `data` using `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]"
j_pivot(ndjson_file, path, as = "data.frame")

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

`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 .

Examples

```
version()
```

Index