Package 'zujson'

November 16, 2026

Title Lightweight JSON Parsing and Serialization

Version 0.1.1

Description Converts between JSON text and ordinary R vectors and lists through a small, predictable set of functions, backed by vendored 'yyjson' <https://github.com/ibireme/yyjson> and requiring no system JSON library. Parsing accepts character, raw and file input and reports failures through structured conditions; serialization writes UTF-8 bytes suitable for use directly as an HTTP request body. The type mapping is deliberately narrow and fully documented, so what goes in and what comes out are both predictable.

License MIT + file LICENSE

URL https://github.com/pedrobtz/zujson, https://pedrobtz.github.io/zujson/

BugReports https://github.com/pedrobtz/zujson/issues

Encoding UTF-8

Language en-GB

Imports utils

Suggests jsonlite, knitr, rmarkdown, testthat (>= 3.2.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph], YaoYuan [ctb, cph] (author of the bundled yyjson library)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

Repository CRAN

Date/Publication 2026-11-16 08:10:02 UTC

Contents

json_parse 2
json_parse_ndjson 5
json_validate 7
json_write 8
zujson_info 10

Index **11**

`json_parse` *Parse JSON into R*

Description

`json_parse()` turns JSON text, or the raw bytes of an HTTP response body, into ordinary R vectors and lists. `json_parse_raw()` and `json_parse_file()` are the same parser reached directly, for when the input type is already known.

Usage

```
json_parse(x, simplify = TRUE, data_frame = FALSE)

json_parse_raw(x, simplify = TRUE, data_frame = FALSE)

json_parse_file(path, simplify = TRUE, data_frame = FALSE)
```

Arguments

`x`	For `json_parse()`, a single string of JSON text or a raw vector of UTF-8 JSON bytes. For `json_parse_raw()`, a raw vector.
`simplify`	How to simplify JSON arrays to atomic vectors: one of `"preserve"` (the default), `"coerce"` or `"none"`. `TRUE` and `FALSE` are accepted as synonyms for `"preserve"` and `"none"`. See *Simplification modes* below.
`data_frame`	Whether an array of objects becomes a data frame. `FALSE` by default, in which case it stays a list of named lists. See *Data frames* below.
`path`	Path to a file containing JSON.

Value

The R object the JSON maps to, by the table above: a JSON object becomes a named `list`, an array an atomic vector when its elements share a kind and a `list` when they do not, a string a length-1 `character`, a number a length-1 `integer` or `double`, `true`/`false` a length-1 `logical`, and `null` becomes `NULL`. The class of the result therefore depends on the JSON and on both arguments: `simplify` decides the mixed-array case, and `data_frame = TRUE` turns an array of objects into a `data.frame`. All three functions return the same value for the same JSON and differ only in where the bytes come from.

Type mapping

JSON objects always become named lists. JSON arrays become an atomic vector when every element agrees on a type and a list otherwise:

JSON	R
`{"a": 1}`	named `list`
`[1, 2, 3]`	`integer`
`[1, 2.5]`	`double`
`[true, false]`	`logical`
`["a", "b"]`	`character`
`[1, "a"]`	`list` (no common type without coercing)
`[1, {}]`	`list` (nested container)
`[]`, `[null, null]`	`logical`
`null`	`NULL`

Numbers become `integer` when they fit in R's 32-bit integer and `double` otherwise. A number too large for any finite `double` becomes `Inf` rather than an error: RFC 8259 sets no limit on the magnitude of a number, so `1e309` is valid JSON. An integer too wide for 64 bits becomes the nearest `double`, the same one the token spelled with `.0` gives. A `null` inside an array being simplified becomes `NA`; a `null` anywhere else becomes `NULL`. Strings arrive as UTF-8.

Simplification modes

`simplify` picks what happens to an array whose elements do not share a kind. The three modes agree everywhere else, including the promotions *within* the numeric family (`[true, 1]` is `c(1L, 1L)` in all of them):

mode	`[1, "a"]` becomes
`"preserve"`, or `TRUE` (default)	`list(1L, "a")`
`"coerce"`	`c("1", "a")`
`"none"`, or `FALSE`	`list(1L, "a")`, and every other array is a list too

`preserve` keeps the type and gives up the uniform shape, on the view that a field which is usually a number and occasionally a string is a bug worth seeing. `coerce` is the opt-in for callers who would rather have the vector: the numbers in an array are gathered into one numeric vector and turned to strings by `as.character()`, `true`/`false` become `"TRUE"`/`"FALSE"`, and `null` stays `NA`. Two consequences follow from using `as.character()`: the strings follow the session's `OutDec` and `scipen` options, and how one number is written depends on the others in the same array – `[100000,"a"]` gives `"100000"` but `[100000,1.5,"a"]` gives `"1e+05"`. A nested object or array is never coerced away.

Data frames

`data_frame = TRUE` turns any non-empty array whose elements are *all* objects into a data frame. Columns are the union of the keys in the order first seen, so the result is rectangular however ragged the records are. A record missing a key contributes `NA` in an atomic column and `NULL` in a list

column – the same as an explicit JSON `null` in either case, because "absent" and "null" are one thing once the value is in a column. Each column is then simplified with the active `simplify` mode, so a column of mixed kinds is a list column under `preserve` and a character column under `coerce`. It applies wherever such an array appears, however deeply nested.

`simplify = "none"` takes precedence: it is the mode that guarantees every JSON array arrives as an R list, and a data frame is not one. The two options are not combined, and `data_frame = TRUE` is ignored under it.

Two limits apply, both raising a structured condition rather than a bare error. An object with the same key twice cannot become a row – a column has one cell per record – so it raises `zujson_parse_error` instead of silently keeping one of the values; plain parsing, which puts both in a list, is unaffected. And because the frame is rectangular, its size is set by the union of the keys rather than by the length of the body: records that share no keys at all would ask for one cell per record *per record*, and no limit on the size of the body can stand in for a limit on that, because the growth is quadratic in it. More than `zujson_info()$max_df_cells` cells raises `zujson_limit_error`. The limit is for the whole document, not for each frame: every data frame in it, nested or side by side, draws on one budget, and `json_parse_ndjson()` counts one body as one document. The default is 50 million – clear of any real tabular response, and far below what a hostile one reaches – and `options(zujson.max_df_cells = )` changes it for the session. `zujson_info()$max_df_cells` reports the limit in force, as a whole number of cells.

Setting it larger than any frame that could be built switches the check off; the option is clamped to what an `R_xlen_t` holds, which is what `zujson_info()$max_df_cells` then reports. Switched off, nothing bounds the allocation but the machine, and a body large enough to exhaust memory raises R's own allocation error rather than a `zujson_error`.

Nesting deeper than 1000 levels is rejected with a `zujson_depth_error`, which is what makes the parser safe to point at an untrusted response body.

A character `x` is read as UTF-8 JSON: text marked UTF-8 or plain ASCII as it is, and text in another encoding converted first. Text that is not valid in the encoding it declares – typically a body read with `readLines()` under the C locale – raises `zujson_parse_error` rather than having its bytes rewritten, and `json_validate()` returns `FALSE` for it. The raw route has no such step: pass the bytes as they arrived.

A leading UTF-8 byte order mark is ignored rather than rejected: RFC 8259 forbids emitting one but allows ignoring it, and real APIs emit them. The bare literals `Infinity`, `-Infinity` and `NaN` are not JSON and stay rejected, which is a separate question from the magnitude of a number that *is* written as one.

Two *further* things are valid JSON and still cannot become R values, and both raise `zujson_parse_error` rather than a bare error, so a caller handling `zujson_error` catches them alongside everything else: a string or key containing an escaped NUL (`\u0000`), which no R string can hold, and one longer than `.Machine$integer.max` bytes. `json_parse_file()` additionally raises `zujson_io_error` when the file cannot be read at all, which is a different problem from its contents not being JSON.

See Also

`json_write()` for the other direction, `json_validate()` to check without building a result.

Examples

```
json_parse('{"ok": true, "ids": [1, 2, 3]}')
```

```
# arrays that have no common type stay lists
json_parse('[1, "a"]')

# raw bytes, as an HTTP response body arrives
json_parse(charToRaw('{"ok": true}'))

# no simplification at all
json_parse('[1, 2, 3]', simplify = FALSE)

# coerce across kinds instead of keeping the type
json_parse('[1, "a"]', simplify = "coerce")

# a number past the range of a double is Inf, not a parse failure
json_parse("[1e309]")

# an array of records, as a data frame
json_parse('[{"id":1,"nm":"a"},{"id":2,"nm":"b"}]', data_frame = TRUE)

json_parse_raw(charToRaw('[1, 2, 3]'))

path <- tempfile(fileext = ".json")
writeLines('{"a": [1, 2]}', path)
json_parse_file(path)
unlink(path)
```

`json_parse_ndjson` *Parse and write NDJSON*

Description

NDJSON (`application/x-ndjson`, also called JSON Lines) is one JSON value per line. `json_parse_ndjson()` turns such a body into a list of records; `json_write_ndjson()` and `json_write_ndjson_raw()` go the other way.

Usage

```
json_parse_ndjson(x, simplify = TRUE, data_frame = FALSE)

json_write_ndjson(x, auto_unbox = TRUE)

json_write_ndjson_raw(x, auto_unbox = TRUE)
```

Arguments

`x`	For `json_parse_ndjson()`, a single string or a raw vector of NDJSON bytes. For the writers, a list of records or a data frame — a data frame is written one object per row, which is the shape NDJSON exists for.
`simplify`	Passed through to `json_parse()` for each record.

`data_frame`	Passed through to `json_parse()` for each record. Note that records are parsed one at a time, so this turns an array *inside* a record into a data frame; it does not make a frame out of the stream.
`auto_unbox`	Passed through to `json_write()` for each record.

Details

Each line is parsed exactly as `json_parse()` would parse it, and each record is written exactly as `json_write()` would write it, so the type mappings documented there apply unchanged. The result of parsing is **always a list**, one element per record, however uniform the records are — records are independent documents, and an NDJSON body is not a table.

Blank lines are skipped rather than treated as records, and `\r\n` line endings are accepted. Every error reports the **line number** – a record that is not JSON, and one that is but cannot become an R value, such as one nested too deep – because "invalid JSON at byte 41827" is not useful in a body of 10,000 records. A UTF-8 byte order mark is ignored at the start of the body and refused anywhere else.

Value

`json_parse_ndjson()` returns a `list` with one element per record, each being what `json_parse()` returns for that line. `json_write_ndjson()` returns a length-1 `character` vector and `json_write_ndjson_raw()` a `raw` vector of UTF-8 bytes; both hold one record per line and end with a trailing newline, so appending another record is always valid.

Why line framing is safe

A raw newline byte cannot appear inside a JSON string — it must be escaped as `\\n` — and zujson's writer always escapes it. A serialized record therefore never contains a bare newline, so splitting on newlines can never cut a record in half. This is the property the format rests on.

It is also why there is no `pretty` argument: indented JSON contains newlines, and a newline inside a record is exactly what NDJSON framing cannot survive. Pretty-printed NDJSON is corrupt, not prettier.

See Also

`json_parse()` and `json_write()` for single documents.

Examples

```
body <- '{"id":1,"ok":true}\n{"id":2,"ok":false}\n'
json_parse_ndjson(body)

# a data frame is one record per row
cat(json_write_ndjson(data.frame(id = 1:2, nm = c("a", "b"))))

# round trip
recs <- list(list(a = 1L), list(b = "x"))
identical(json_parse_ndjson(json_write_ndjson(recs)), recs)

# bytes, ready to be an HTTP request body
```

```
json_write_ndjson_raw(list(list(a = 1L), list(a = 2L)))
```

`json_validate` *Check whether input is valid JSON*

Description

Parses `x` and reports whether it succeeded, without building an R result and without raising a condition. Use it to decide whether a response body is worth parsing; use `json_parse()` when a failure should be an error you can read.

Usage

```
json_validate(x)
```

Arguments

`x`	A single string of JSON text, or a raw vector of JSON bytes.

Details

This answers "is this valid JSON", which is very nearly but not exactly "will `json_parse()` succeed". The two come apart on JSON that is valid and still cannot become an R value, of which there are two kinds:

- a string or key R cannot hold: one containing an escaped NUL (`\u0000`), or one longer than `.Machine$integer.max` bytes. `json_parse()` raises `zujson_parse_error`.
- nesting deeper than 1000 levels. RFC 8259 leaves any depth limit to the implementation, so such a document is still valid JSON. `json_parse()` raises `zujson_depth_error`, because its recursive tree builder has a C stack to protect; the reader underneath it is iterative and does not.

This function returns `TRUE` for both, and is right to: it is asked whether the bytes are JSON, not whether this package can materialise them. Code that must not fail should handle the condition from `json_parse()` rather than pre-screening with this.

Value

A length-1 `logical`, never `NA`: `TRUE` when `x` is well-formed JSON by the reader's rules, `FALSE` otherwise. `NA_character_` is `FALSE` rather than `NA`. A `TRUE` is not a promise that `json_parse()` will succeed – the two cases above are valid JSON that this package cannot turn into an R value.

Examples

```
json_validate('{"a": 1}')
json_validate('{"a": }')
json_validate(charToRaw("[]"))
```

`json_write` *Serialize R as JSON*

Description

`json_write()` renders an R object as JSON text. `json_write_raw()` returns the same JSON as UTF-8 bytes, which is the form an HTTP request body wants.

Usage

```
json_write(x, pretty = FALSE, auto_unbox = TRUE)

json_write_raw(x, pretty = FALSE, auto_unbox = TRUE)
```

Arguments

`x`	The R object to serialize.
`pretty`	Whether to indent the output. `FALSE` (the default) writes the compact form, which is what you want for a request body.
`auto_unbox`	Whether a length-1 atomic vector becomes a bare JSON scalar (`TRUE`, the default) or a one-element array. Wrap a value in `I()` to keep it an array under `auto_unbox = TRUE`; that is the escape hatch for an API field that must always be a list.

Value

`json_write()` returns a length-1 `character` vector holding the JSON text. `json_write_raw()` returns the same document as a `raw` vector of UTF-8 bytes, which is what an HTTP request body wants, so that direction needs no conversion step.

Type mapping

R	JSON
`NULL`	`null`
`NA`, `NaN`, `Inf`	`null`
`list(a = 1, b = 2)`	`{"a":1,"b":2}`
`list(1, 2)`	`[1,2]`
`c(a = 1, b = 2)`	`{"a":1,"b":2}`
`1:3`	`[1,2,3]`
`"a"`	`"a"` (see `auto_unbox`)
`I("a")`	`["a"]`
`factor`	its level, as a string
`Date`	`"YYYY-MM-DD"`
`POSIXct`	`"YYYY-MM-DDTHH:MM:SSZ"`, in UTC
`bit64::integer64`	the integers it holds
`data.frame`	array of one object per row

```
list()                                  []
structure(list(), names = character())  {}
```

A vector or list becomes a JSON **object** when every element is named, and an **array** otherwise. Partial names would produce keys like `""`, which is valid JSON but almost never intended, so a partially named vector is written as an array and its names are dropped. That includes a JSON object with an empty key, `{"":1}`: it parses to a list named `""`, which is written back as an array, `[1]`.

Data frames are written row-oriented, because that is what an HTTP API means by a table. To get the column-oriented form instead, strip the class first: `json_write(as.list(df))`.

Every kind of missing value becomes `null`: JSON has no `NA`, and `NaN` and `Infinity` are not JSON either. `POSIXct` is always written as UTC no matter what its `tzone` says, since it is the same instant either way, and sub-second parts are dropped. Both are doubles in R and so reach much further than a timestamp can be written: outside `0000-01-01` to `9999-12-31`, the four-digit years ISO 8601 allows, they raise a `zujson_write_error` rather than print a year no API can parse. A `Date` or `POSIXct` that is all `NA` but stored as logical, which is how `structure(NA, class = "Date")` comes out, is written as `null`; one stored as anything else that is not a number raises `zujson_unsupported_type`.

An `integer64` from bit64 is written as the 64-bit integers it holds, with its `NA` as `null`. Its values live in the bits of a double, so writing the double would send `1` as `5e-324`. bit64 itself is not needed to write one.

A string whose `Encoding()` is `"bytes"` also raises a `zujson_write_error`. That marking is R saying it does not know the encoding, and JSON text is UTF-8 by definition, so there is nothing to convert from. Every other encoding R tracks is converted on the way out, and a string that is not valid in the encoding it declares raises `zujson_write_error` rather than being written with its bytes rewritten.

Doubles are written compactly and always read back as the same number, so `0.1` is `0.1` rather than `0.10000000000000001`. A double that is a whole number is written without a decimal point at all — R has no integer literal, so `1` is a double, and `1.0` is rejected by a schema expecting an integer.

Complex vectors, raw vectors, functions, environments and `POSIXlt` have no sensible JSON form and raise a `zujson_unsupported_type` error rather than being guessed at, wherever they appear: as a value, a list element, a data frame column or an NDJSON record. (`POSIXlt` is a list of 11 broken-down time fields; convert it with `as.POSIXct()` first.) So does a classed vector whose storage the class cannot mean, such as a factor whose levels are not character. A vector carrying an unrecognised class is written as its underlying type, which for a matrix means its values in column-major order with `dim` dropped — matrices are not turned into nested arrays in this version.

See Also

`json_parse()` for the other direction.

Examples

```
json_write(list(name = "ada", ids = 1:3, ok = TRUE))

# length-1 vectors unbox by default; I() keeps them arrays
```

```
json_write(list(tag = "x", tags = I("x")))

# every atomic vector stays an array
json_write(list(tag = "x"), auto_unbox = FALSE)

cat(json_write(list(a = 1, b = list(c = 2)), pretty = TRUE))

json_write(data.frame(id = 1:2, nm = c("a", "b")))

# bytes, ready to be an HTTP request body
json_write_raw(list(q = "search"))
```

`zujson_info` *Report what this build of zujson contains*

Description

Returns the package version, the version of the vendored yyjson sources it was compiled against, and the nesting depth limit the parser and serializer both enforce.

Usage

```
zujson_info()
```

Value

A named list with `zujson`, `yyjson`, `max_depth` and `max_df_cells`. `max_depth` is compiled in and fixed; `max_df_cells` is the limit in force, which `options(zujson.max_df_cells = )` changes.

Examples

```
zujson_info()
```

Index

I(), *8*

json_parse, 2
json_parse(), *5–7*, *9*
json_parse_file *(*json_parse*)*, 2
json_parse_ndjson, 5
json_parse_raw *(*json_parse*)*, 2
json_validate, 7
json_validate(), *4*
json_write, 8
json_write(), *4*, *6*
json_write_ndjson *(*json_parse_ndjson*)*, 5
json_write_ndjson_raw
        *(*json_parse_ndjson*)*, 5
json_write_raw *(*json_write*)*, 8

zujson_info, 10