Package 'jsonlite'

April 8, 2017

Version 1.4

Title A Robust, High Performance JSON Parser and Generator for R

License MIT + file LICENSE

NeedsCompilation yes

Depends methods

Author Jeroen Ooms, Duncan Temple Lang, Lloyd Hilaiel

URL https://arxiv.org/abs/1403.2805,
https://www.opencpu.org/posts/jsonlite-a-smarter-json-encoder

BugReports http://github.com/jeroen/jsonlite/issues

Maintainer Jeroen Ooms <jeroen@berkeley.edu>

VignetteBuilder knitr, R.rsp

Description A fast JSON parser and generator optimized for statistical data and the web. Started out as a fork of 'RJSONIO', but has been completely rewritten in recent versions. The package offers flexible, robust, high performance tools for working with JSON in R and is particularly powerful for building pipelines and interacting with a web API. The implementation is based on the mapping described in the vignette (Ooms, 2014). In addition to converting JSON data from/to R objects, 'jsonlite' contains functions to stream, validate, and prettify JSON data. The unit tests included with the package verify that all edge cases are encoded and decoded consistently for use with dynamic data in systems and applications.

Suggests httr, curl, plyr, testthat, knitr, rmarkdown, R.rsp, sp

RoxygenNote 6.0.1

Repository CRAN

Date/Publication 2017-04-08 21:03:40 UTC

R topics documented:

`base64` *Encode/decode base64*

Description

Simple in-memory base64 encoder and decoder. Used internally for converting raw vectors to text. Interchangable with encoder from `base64enc` or `openssl` package.

Usage

```
base64_dec(input)

base64_enc(input)
```

Arguments

`input`	string or raw vector to be encoded/decoded

Examples

```
str <- base64_enc(serialize(iris, NULL))
out <- unserialize(base64_dec(str))
stopifnot(identical(out, iris))
```

`flatten` *Flatten nested data frames*

Description

In a nested data frame, one or more of the columns consist of another data frame. These structures frequently appear when parsing JSON data from the web. We can flatten such data frames into a regular 2 dimensional tabular structure.

Usage

```
flatten(x, recursive = TRUE)
```

Arguments

x	a data frame
recursive	flatten recursively

Examples

```
options(stringsAsFactors=FALSE)
x <- data.frame(driver = c("Bowser", "Peach"), occupation = c("Koopa", "Princess"))
x$vehicle <- data.frame(model = c("Piranha Prowler", "Royal Racer"))
x$vehicle$stats <- data.frame(speed = c(55, 34), weight = c(67, 24), drift = c(35, 32))
str(x)
str(flatten(x))
str(flatten(x, recursive = FALSE))

## Not run:
data1 <- fromJSON("https://api.github.com/users/hadley/repos")
colnames(data1)
colnames(data1$owner)
colnames(flatten(data1))

# or for short:
data2 <- fromJSON("https://api.github.com/users/hadley/repos", flatten = TRUE)
colnames(data2)

## End(Not run)
```

---

prettify, minify *Prettify or minify a JSON string*

Description

Prettify adds indentation to a JSON string; minify removes all indentation/whitespace.

Usage

```
prettify(txt, indent = 4)

minify(txt)
```

Arguments

txt	JSON string
indent	number of spaces to indent

Examples

```
myjson <- toJSON(cars)
cat(myjson)
prettify(myjson)
minify(myjson)
```

`rbind.pages` *Combine pages into a single data frame*

Description

The `rbind.pages` function is used to combine a list of data frames into a single data frame. This is often needed when working with a JSON API that limits the amount of data per request. If we need more data than what fits in a single request, we need to perform multiple requests that each retrieve a fragment of data, not unlike pages in a book. In practice this is often implemented using a `page` parameter in the API. The `rbind.pages` function can be used to combine these pages back into a single dataset.

Usage

```
## S3 method for class 'pages'
rbind(pages)
```

Arguments

`pages`	a list of data frames, each representing a *page* of data

Details

The `rbind.pages` function generalizes `base::rbind` and `plyr::rbind.fill` with added support for nested data frames. Not each column has to be present in each of the individual data frames; missing columns will be filled up in `NA` values.

Examples

```
# Basic example
x <- data.frame(foo = rnorm(3), bar = c(TRUE, FALSE, TRUE))
y <- data.frame(foo = rnorm(2), col = c("blue", "red"))
rbind.pages(list(x, y))

## Not run:
baseurl <- "http://projects.propublica.org/nonprofits/api/v1/search.json"
pages <- list()
for(i in 0:20){
  mydata <- fromJSON(paste0(baseurl, "?order=revenue&sort_order=desc&page=", i))
  message("Retrieving page ", i)
  pages[[i+1]] <- mydata$filings
}
```

```
filings <- rbind.pages(pages)
nrow(filings)
colnames(filings)

## End(Not run)
```

`read_json` *Read/write JSON*

Description

Convenience wrappers around toJSON and fromJSON to read and write directly to/from disk.

Usage

```
read_json(path, simplifyVector = FALSE, ...)

write_json(x, path, ...)
```

Arguments

`path`	file on disk
`simplifyVector`	simplifies nested lists into vectors and data frames. See fromJSON.
`...`	additional arguments passed to toJSON or fromJSON
`x`	an object to be serialized to JSON

Examples

```
tmp <- tempfile()
write_json(iris, tmp)

# Nested lists
read_json(tmp)

# A data frame
read_json(tmp, simplifyVector = TRUE)
```

`serializeJSON` *serialize R objects to JSON*

Description

The `serializeJSON` and `unserializeJSON` functions convert between R objects to JSON data. Instead of using a class based mapping like `toJSON` and `fromJSON`, the serialize functions base the encoding schema on the storage type, and capture all data and attributes from any object. Thereby the object can be restored almost perfectly from its JSON representation, but the resulting JSON output is very verbose. Apart from environments, all standard storage types are supported.

Usage

```
serializeJSON(x, digits = 8, pretty = FALSE)

unserializeJSON(txt)
```

Arguments

`x`	an R object to be serialized
`digits`	max number of digits (after the dot) to print for numeric values
`pretty`	add indentation/whitespace to JSON output. See `prettify`
`txt`	a JSON string which was created using `serializeJSON`

Note

JSON is a text based format which leads to loss of precision when printing numbers.

Examples

```
jsoncars <- serializeJSON(mtcars)
mtcars2 <- unserializeJSON(jsoncars)
identical(mtcars, mtcars2)

set.seed('123')
myobject <- list(
  mynull = NULL,
  mycomplex = lapply(eigen(matrix(-rnorm(9),3)), round, 3),
  mymatrix = round(matrix(rnorm(9), 3),3),
  myint = as.integer(c(1,2,3)),
  mydf = cars,
  mylist = list(foo='bar', 123, NA, NULL, list('test')),
  mylogical = c(TRUE,FALSE,NA),
  mychar = c('foo', NA, 'bar'),
  somemissings = c(1,2,NA,NaN,5, Inf, 7 -Inf, 9, NA),
  myrawvec = charToRaw('This is a test')
);
identical(unserializeJSON(serializeJSON(myobject)), myobject);
```

`stream_in, stream_out` *Streaming JSON input/output*

Description

The `stream_in` and `stream_out` functions implement line-by-line processing of JSON data over a `connection`, such as a socket, url, file or pipe. JSON streaming requires the ndjson format, which slightly differs from `fromJSON` and `toJSON`, see details.

Usage

```
stream_in(con, handler = NULL, pagesize = 500, verbose = TRUE, ...)

stream_out(x, con = stdout(), pagesize = 500, verbose = TRUE,
  prefix = "", ...)
```

Arguments

`con`	a `connection` object. If the connection is not open, `stream_in` and `stream_out` will automatically open and later close (and destroy) the connection. See details.
`handler`	a custom function that is called on each page of JSON data. If not specified, the default handler stores all pages and binds them into a single data frame that will be returned by `stream_in`. See details.
`pagesize`	number of lines to read/write from/to the connection per iteration.
`verbose`	print some information on what is going on.
`...`	arguments for `fromJSON` and `toJSON` that control JSON formatting/parsing where applicable. Use with caution.
`x`	object to be streamed out. Currently only data frames are supported.
`prefix`	string to write before each line (use `"\u001e"` to write rfc7464 text sequences)

Details

Because parsing huge JSON strings is difficult and inefficient, JSON streaming is done using **lines of minified JSON records**, a.k.a. ndjson. This is pretty standard: JSON databases such as dat or MongoDB use the same format to import/export datasets. Note that this means that the total stream combined is not valid JSON itself; only the individual lines are. Also note that because line-breaks are used as separators, prettified JSON is not permitted: the JSON lines *must* be minified. In this respect, the format is a bit different from `fromJSON` and `toJSON` where all lines are part of a single JSON structure with optional line breaks.

The `handler` is a callback function which is called for each page (batch) of JSON data with exactly one argument (usually a data frame with `pagesize` rows). If `handler` is missing or `NULL`, a default handler is used which stores all intermediate pages of data, and at the very end binds all pages together into one single data frame that is returned by `stream_in`. When a custom `handler` function is specified, `stream_in` does not store any intermediate results and always returns `NULL`. It is then up to the `handler` to process or store data pages. A `handler` function that does not store intermediate

results in memory (for example by writing output to another connection) results in a pipeline that can process an unlimited amount of data. See example.

If a connection is not opened yet, `stream_in` and `stream_out` will automatically open and later close the connection. Because R destroys connections when they are closed, they cannot be reused. To use a single connection for multiple calls to `stream_in` or `stream_out`, it needs to be opened beforehand. See example.

Value

The `stream_out` function always returns `NULL`. When no custom handler is specified, `stream_in` returns a data frame of all pages binded together. When a custom handler function is specified, `stream_in` always returns `NULL`.

References

MongoDB export format: http://docs.mongodb.org/manual/reference/program/mongoexport/#cmdoption--query

Documentation for the JSON Lines text file format: http://jsonlines.org/

Examples

```
# compare formats
x <- iris[1:3,]
toJSON(x)
stream_out(x)

# Trivial example
mydata <- stream_in(url("http://httpbin.org/stream/100"))

## Not run: stream large dataset to file and back
library(nycflights13)
stream_out(flights, file(tmp <- tempfile()))
flights2 <- stream_in(file(tmp))
unlink(tmp)
all.equal(flights2, as.data.frame(flights))

# stream over HTTP
diamonds2 <- stream_in(url("http://jeroen.github.io/data/diamonds.json"))

# stream over HTTP with gzip compression
flights3 <- stream_in(gzcon(url("http://jeroen.github.io/data/nycflights13.json.gz")))
all.equal(flights3, as.data.frame(flights))

# stream over HTTPS (HTTP+SSL) via curl
library(curl)
flights4 <- stream_in(gzcon(curl("https://jeroen.github.io/data/nycflights13.json.gz")))
all.equal(flights4, as.data.frame(flights))

# or alternatively:
flights5 <- stream_in(gzcon(pipe("curl https://jeroen.github.io/data/nycflights13.json.gz")))
all.equal(flights5, as.data.frame(flights))
```

```
# Full JSON IO stream from URL to file connection.
# Calculate delays for flights over 1000 miles in batches of 5k
library(dplyr)
con_in <- gzcon(url("http://jeroen.github.io/data/nycflights13.json.gz"))
con_out <- file(tmp <- tempfile(), open = "wb")
stream_in(con_in, handler = function(df){
  df <- dplyr::filter(df, distance > 1000)
  df <- dplyr::mutate(df, delta = dep_delay - arr_delay)
  stream_out(df, con_out, pagesize = 1000)
}, pagesize = 5000)
close(con_out)

# stream it back in
mydata <- stream_in(file(tmp))
nrow(mydata)
unlink(tmp)

# Data from http://openweathermap.org/current#bulk
# Each row contains a nested data frame.
daily14 <- stream_in(gzcon(url("http://78.46.48.103/sample/daily_14.json.gz")), pagesize=50)
subset(daily14, city$name == "Berlin")$data[[1]]

# Or with dplyr:
library(dplyr)
daily14f <- flatten(daily14)
filter(daily14f, city.name == "Berlin")$data[[1]]

# Stream import large data from zip file
tmp <- tempfile()
download.file("http://jsonstudio.com/wp-content/uploads/2014/02/companies.zip", tmp)
companies <- stream_in(unz(tmp, "companies.json"))

## End(Not run)
```

`toJSON, fromJSON` *Convert* R *objects to/from JSON*

Description

These functions are used to convert between JSON data and R objects. The `toJSON` and `fromJSON` functions use a class based mapping, which follows conventions outlined in this paper: https://arxiv.org/abs/1403.2805 (also available as vignette).

Usage

```
fromJSON(txt, simplifyVector = TRUE, simplifyDataFrame = simplifyVector,
  simplifyMatrix = simplifyVector, flatten = FALSE, ...)

toJSON(x, dataframe = c("rows", "columns", "values"), matrix = c("rowmajor",
```

```
"columnmajor"), Date = c("ISO8601", "epoch"), POSIXt = c("string",
"ISO8601", "epoch", "mongo"), factor = c("string", "integer"),
complex = c("string", "list"), raw = c("base64", "hex", "mongo"),
null = c("list", "null"), na = c("null", "string"), auto_unbox = FALSE,
digits = 4, pretty = FALSE, force = FALSE, ...)
```

Arguments

txt	a JSON string, URL or file
simplifyVector	coerce JSON arrays containing only primitives into an atomic vector
simplifyDataFrame	coerce JSON arrays containing only records (JSON objects) into a data frame
simplifyMatrix	coerce JSON arrays containing vectors of equal mode and dimension into matrix or array
flatten	automatically flatten nested data frames into a single non-nested data frame
...	arguments passed on to class specific print methods
x	the object to be encoded
dataframe	how to encode data.frame objects: must be one of 'rows', 'columns' or 'values'
matrix	how to encode matrices and higher dimensional arrays: must be one of 'rowmajor' or 'columnmajor'.
Date	how to encode Date objects: must be one of 'ISO8601' or 'epoch'
POSIXt	how to encode POSIXt (datetime) objects: must be one of 'string', 'ISO8601', 'epoch' or 'mongo'
factor	how to encode factor objects: must be one of 'string' or 'integer'
complex	how to encode complex numbers: must be one of 'string' or 'list'
raw	how to encode raw objects: must be one of 'base64', 'hex' or 'mongo'
null	how to encode NULL values within a list: must be one of 'null' or 'list'
na	how to print NA values: must be one of 'null' or 'string'. Defaults are class specific
auto_unbox	automatically unbox all atomic vectors of length 1. It is usually safer to avoid this and instead use the unbox function to unbox individual elements. An exception is that objects of class AsIs (i.e. wrapped in I()) are not automatically unboxed. This is a way to mark single values as length-1 arrays.
digits	max number of decimal digits to print for numeric values. Use I() to specify significant digits. Use NA for max precision.
pretty	adds indentation whitespace to JSON output. Can be TRUE/FALSE or a number specifying the number of spaces to indent. See prettify
force	unclass/skip objects of classes with no defined JSON mapping

Details

The toJSON and fromJSON functions are drop-in replacements for the identically named functions in packages rjson and RJSONIO. Our implementation uses an alternative, somewhat more consistent mapping between R objects and JSON strings.

The serializeJSON and unserializeJSON functions in this package use an alternative system to convert between R objects and JSON, which supports more classes but is much more verbose.

A JSON string is always unicode, using UTF-8 by default, hence there is usually no need to escape any characters. However, the JSON format does support escaping of unicode characters, which are encoded using a backslash followed by a lower case "u" and 4 hex characters, for example: "Z\u00FCrich". The fromJSON function will parse such escape sequences but it is usually preferable to encode unicode characters in JSON using native UTF-8 rather than escape sequences.

References

Jeroen Ooms (2014). The jsonlite Package: A Practical and Consistent Mapping Between JSON Data and R Objects. *arXiv:1403.2805*. https://arxiv.org/abs/1403.2805

Examples

```
# Stringify some data
jsoncars <- toJSON(mtcars, pretty=TRUE)
cat(jsoncars)

# Parse it back
fromJSON(jsoncars)

# Parse escaped unicode
fromJSON('{"city" : "Z\\u00FCrich"}')

# Decimal vs significant digits
toJSON(pi, digits=3)
toJSON(pi, digits=I(3))

## Not run: retrieve data frame
data1 <- fromJSON("https://api.github.com/users/hadley/orgs")
names(data1)
data1$login

# Nested data frames:
data2 <- fromJSON("https://api.github.com/users/hadley/repos")
names(data2)
names(data2$owner)
data2$owner$login

# Flatten the data into a regular non-nested dataframe
names(flatten(data2))

# Flatten directly (more efficient):
data3 <- fromJSON("https://api.github.com/users/hadley/repos", flatten = TRUE)
identical(data3, flatten(data2))
```

```
## End(Not run)
```

---

unbox *Unbox a vector or data frame*

Description

This function marks an atomic vector or data frame as a singleton, i.e. a set with exactly 1 element. Thereby, the value will not turn into an `array` when encoded into JSON. This can only be done for atomic vectors of length 1, or data frames with exactly 1 row. To automatically unbox all vectors of length 1 within an object, use the `auto_unbox` argument in `toJSON`.

Usage

```
unbox(x)
```

Arguments

x	atomic vector of length 1, or data frame with 1 row.

Details

It is usually recommended to avoid this function and stick with the default encoding schema for the various R classes. The only use case for this function is if you are bound to some specific predefined JSON structure (e.g. to submit to an API), which has no natural R representation. Note that the default encoding for data frames naturally results in a collection of key-value pairs, without using unbox.

Value

Returns a singleton version of `x`.

References

http://en.wikipedia.org/wiki/Singleton_(mathematics)

Examples

```
toJSON(list(foo=123))
toJSON(list(foo=unbox(123)))

# Auto unbox vectors of length one:
x = list(x=1:3, y = 4, z = "foo", k = NULL)
toJSON(x)
toJSON(x, auto_unbox = TRUE)

x <- iris[1,]
toJSON(list(rec=x))
toJSON(list(rec=unbox(x)))
```

validate *Validate JSON*

Description

Test if a string contains valid JSON. Characters vectors will be collapsed into a single string.

Usage

```
validate(txt)
```

Arguments

txt	JSON string

Examples

```
#Output from toJSON and serializeJSON should pass validation
myjson <- toJSON(mtcars)
validate(myjson) #TRUE

#Something bad happened
truncated <- substring(myjson, 1, 100)
validate(truncated) #FALSE
```

Index