Package 'tidyjson'

April 21, 2017

Title A Grammar for Turning 'JSON' into Tidy Tables

Version 0.2.2

Author Jeremy Stanley <jeremy.stanley@gmail.com>

Maintainer ORPHANED

Description An easy and consistent way to turn 'JSON' into tidy data frames that are natural to work with in 'dplyr', 'ggplot2' and other tools.

Depends R (>= 3.1.0)

Imports assertthat, jsonlite, dplyr

URL https://github.com/sailthru/tidyjson

License MIT + file LICENSE

LazyData true

Suggests knitr, testthat, ggplot2, magrittr

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2017-04-21 14:55:49 UTC

X-CRAN-Original-Maintainer Jeremy Stanley <jeremy.stanley@gmail.com>

X-CRAN-Comment Orphaned and corrected on 2017-04-21 as check errors were not corrected despite reminders.

R topics documented:

allowed_json_types 2
append_values 3
append_values_factory 3
append_values_type 4
commits 4
companies 5
determine_types 6
enter_object 6

gather_array 7
gather_keys 8
issues 9
jfactory 10
jfunctions 10
json_lengths 11
json_types 11
list_path 12
my_unlist 12
prep_path 13
read_json 13
replace_nulls 14
spread_values 14
tbl_json 15
tidyjson 15
worldbank 16
wrap_dplyr_verb 17
[.tbl_json 17

Index **18**

`allowed_json_types` *Fundamental JSON types from http://json.org/, where I collapse 'true' and 'false' into 'logical'*

Description

Fundamental JSON types from http://json.org/, where I collapse 'true' and 'false' into 'logical'

Usage

```
allowed_json_types
```

Format

```
chr [1:6] "object" "array" "string" "number" "logical" "null"
```

append_values *Appends all values with a specified type as a new column*

Description

The append_values_X functions let you take any remaining JSON and add it as a column X (for X in "string", "number", "logical") insofar as it is of the JSON type specified.

Usage

```
append_values_string(x, column.name = type, force = TRUE)

append_values_number(x, column.name = type, force = TRUE)

append_values_logical(x, column.name = type, force = TRUE)
```

Arguments

x	a tbl_json object
column.name	the column.name to append the values into the data.frame under
force	parameter that determines if the variable type should be computed or not if force is FALSE, then the function may take more memory

Details

Any values that do not conform to the type specified will be NA in the resulting column. This includes other scalar types (e.g., numbers or logicals if you are using append_values_string) and *also* any rows where the JSON is still an object or an array.

Examples

```
library(magrittr)  # for %>%
'{"first": "bob", "last": "jones"}' %>%
  gather_keys() %>%
  append_values_string()
```

append_values_factory *Creates the append_values_* functions*

Description

Creates the append_values_* functions

Usage

```
append_values_factory(type, as.value)
```

Arguments

type	the JSON type that will be appended
as.value	function to force coercion to numeric, string, or logical

---

append_values_type *get list of values from json*

Description

get list of values from json

Usage

```
append_values_type(json, type)
```

Arguments

json	extracted using attributes
type	input type (numeric, string, etc)

---

commits *Commit data for the dplyr repo from github API*

Description

Commit data for the dplyr repo from github API

Usage

```
commits
```

Format

JSON

Examples

```
library(dplyr)

# Show first 200 characters of JSON
commits %>% substr(1, 200) %>% writeLines

# Extract metadata for every commit
commits %>%   # single json document of github commits from dplyr
  gather_array %>%  # stack as an array
  spread_values(
    sha         = jstring("sha"),
    author      = jstring("commit", "author", "name"),
    author.date = jstring("commit", "author", "date")
  ) %>% head
```

---

companies *Startup company information for 1,000 companies*

Description

From: http://jsonstudio.com/resources/

Usage

```
companies
```

Format

JSON

Examples

```
library(dplyr)
library(jsonlite)

# Print the first record (do not run automatically)
# companies[[1]] %>% prettify

# Get the key employees data
key_employees <- companies %>%
  spread_values(
    name = jstring("name")
  ) %>%
  mutate(
    company.sort_order = rank(name)
  ) %>%
  enter_object("relationships") %>%
  gather_array("relationship.index") %>%
  spread_values(
```

```
    is.past = jlogical("is_past"),
    name = jstring("person", "permalink"),
    title = jstring("title")
  )

# Show the top 10 titles
key_employees %>%
  filter(!is.past) %>%
  group_by(title) %>%
  tally() %>%
  arrange(desc(n)) %>%
  top_n(10)
```

`determine_types` *Determines the types of a list of parsed JSON*

Description

Determines the types of a list of parsed JSON

Usage

```
determine_types(json_list)
```

Arguments

`json_list`	a list of parsed JSON

Value

a factor with levels json_types

`enter_object` *Dive into a specific object "key"*

Description

JSON can contain nested objects, such as "key1": "key2": [1, 2, 3]. The function enter_object() can be used to access the array nested under "key1" and "key2". After using enter_object(), all further tidyjson calls happen inside the referenced object (all other JSON data outside the object is discarded). If the object doesn't exist for a given row / index, then that data.frame row will be discarded.

Usage

```
enter_object(x, ...)
```

Arguments

x	a tbl_json object
...	path to filter

Details

This is useful when you want to limit your data to just information found in a specific key. Use the ... to specific a sequence of keys that you want to enter into. Keep in mind that any rows with JSON that do not contain the key will be discarded by this function.

Examples

```
library(magrittr)  # for %>%
c('{"name": "bob", "children": ["sally", "george"]}', '{"name": "anne"}') %>%
  spread_values(parent.name = jstring("name")) %>%
  enter_object("children") %>%
  gather_array %>%
  append_values_string("children")
```

---

gather_array *Stack a JSON array*

Description

Given a JSON array, such as [1, 2, 3], gather_array will "stack" the array in the tbl_json data.frame, by replicating each row of the data.frame by the length of the corresponding JSON array. A new column (by default called "array.index") will be added to keep track of the referenced position in the array for each row of the resuling data.frame.

Usage

```
gather_array(x, column.name = "array.index")
```

Arguments

x	a tbl_json whose JSON attribute should always be an array
column.name	the name to give to the array index column created

Details

JSON can contain arrays of data, which can be simple vectors (fixed or varying length integer, character or logical vectors). But they also often contain lists of other objects (like a list of purchases for a user). The function gather_array() takes JSON arrays and duplicates the rows in the data.frame to correspond to the indices of the array, and puts the elements of the array into the JSON attribute. This is equivalent to "stacking" the array in the data.frame, and lets you continue to manipulate the remaining JSON in the elements of the array. For simple arrays, use append_values_* to capture all of the values of the array. For more complex arrays (where the values are themselves objects or arrays), continue using other tidyjson functions to structure the data as needed.

Value

a tbl_json with a new column (column.name) that captures the array index and JSON attribute extracted from the array

Examples

```
library(magrittr)  # for %>%
'[1, "a", {"k": "v"}]' %>% gather_array %>% json_types
```

`gather_keys` *Stack a JSON "key": value object*

Description

Given a JSON key value structure, like "key1": 1, "key2": 2, the gather_keys() function duplicates the rows of the tbl_json data.frame for every key, adds a new column (default name "key") to capture the key names, and then dives into the JSON values to enable further manipulation with downstream tidyjson functions.

Usage

```
gather_keys(x, column.name = "key")
```

Arguments

`x`	a tbl_json whose JSON attribute should always be an object
`column.name`	the name to give to the column of key names created

Details

This allows you to *enter into* the keys of the objects just like ‘gather_array‘ let you enter elements of the array.

Value

a tbl_json with a new column (column.name) that captures the keys and JSON attribute of the associated value data

Examples

```
library(magrittr)  # for %>%
'{"name": "bob", "age": 32}' %>% gather_keys %>% json_types
```

issues *Issue data for the dplyr repo from github API*

Description

Issue data for the dplyr repo from github API

Usage

```
issues
```

Format

JSON

Examples

```
library(dplyr)

# Show first 200 characters of JSON
issues %>% substr(1, 200) %>% writeLines

# Extract metadata for every issue
issues %>%          # single json document of github issues from dplyr
  gather_array %>%  # stack as an array
  spread_values(
    id          = jnumber("id"),
    number      = jnumber("number"),
    title       = jstring("title"),
    user.login  = jstring("user", "login"),
    sate        = jstring("state"),
    locked      = jlogical("locked"),
    comments    = jnumber("comments")
  ) %>% head

# Extract label content for issues with labels
issues %>%          # single json document of github issues from dplyr
  gather_array %>%  # stack as an array
  spread_values(id = jnumber("id")) %>% # capture issue id for relational purposes
  enter_object("labels") %>%            # filter just those with labels
  gather_array("label.index") %>%       # stack labels
  spread_values(
    url   = jstring("url"),
    name  = jstring("name"),
    color = jstring("color")
  ) %>% head

# Get all URLs at the top level of the JSON
issues %>%
  gather_array %>%
```

```
  gather_keys %>%
  append_values_string() %>%
  filter(grepl("url", key)) %>%
  head
```

`jfactory` *Factory that creates the j* functions below*

Description

Factory that creates the j* functions below

Usage

```
jfactory(na.value, conversion.function)
```

Arguments

`na.value`	value to replace NULL with
`conversion.function`	function to convert vector to appropriate type

`jfunctions` *Navigates nested objects to get at keys of a specific type, to be used as arguments to spread_values*

Description

Navigates nested objects to get at keys of a specific type, to be used as arguments to spread_values

Usage

```
jstring(...)

jnumber(...)

jlogical(...)
```

Arguments

`...`	the path to follow

Value

a function that can operate on parsed JSON data

json_lengths *Add a column that contains the length of the JSON data*

Description

When investigating JSON data it can be helpful to identify the lengths of the JSON objects or arrays, especialy when they are 'ragged' across documents. The json_lengths() function adds a column (default name "length") that contains the 'length' of the JSON associated with each row. For objects, this will be equal to the number of keys. For arrays, this will be equal to the length of the array. All scalar values will be of length 1.

Usage

```
json_lengths(x, column.name = "length")
```

Arguments

x	a tbl_json object
column.name	the name to specify for the length column

Value

a tbl_json object with column.name column that tells the length

Examples

```
library(magrittr)  # for %>%
c('[1, 2, 3]', '{"k1": 1, "k2": 2}', '1', {}) %>% json_lengths
```

json_types *Add a column that tells the 'type' of the data in the root of the JSON*

Description

The function json_types() inspects the JSON associated with each row of the tbl_json data.frame, and adds a new column ("type" by default) that identifies the type according to the JSON standard at http://json.org/.

Usage

```
json_types(x, column.name = "type")
```

Arguments

x	a tbl_json object
column.name	the name to specify for the type column

Details

This is particularly useful for inspecting your JSON data types, and can added after gather_array() (or gather_keys()) to inspect the types of the elements (or values) in arrays (or objects).

Value

a tbl_json object with column.name column that tells the type

Examples

```
library(magrittr)  # for %>%
c('{"a": 1}', '[1, 2]', '"a"', '1', 'true', 'null') %>% json_types
```

---

list_path *Recursively access a path*

Description

Recursively access a path

Usage

```
list_path(l, path)
```

Arguments

l	a list
path	a path of keys to follow

---

my_unlist *Unlists while preserving NULLs and only unlisting lists with one value*

Description

Unlists while preserving NULLs and only unlisting lists with one value

Usage

```
my_unlist(l)
```

Arguments

l	a list that we want to unlist

prep_path *Prepare a path from ...*

Description

Prepare a path from ...

Usage

```
prep_path(...)
```

Arguments

...	character strings to construct into a path

read_json *Reads JSON from an input uri (file, url, ...) and returns a tbl_json*

Description

Reads JSON from an input uri (file, url, ...) and returns a tbl_json

Usage

```
read_json(path, format = c("json", "jsonl", "infer"))
```

Arguments

path	to some json data
format	If "json", process the data like one large JSON record. If "jsonl", process the data one JSON record per line (json lines format) If "infer", the format is the suffix of the given filepath.

Value

tbl_json instance

replace_nulls *Replace nulls with something else*

Description

Replace nulls with something else

Usage

```
replace_nulls(l, replace)
```

Arguments

l	a list
replace	what to replace NULLs in the list with

spread_values *Create new columns with JSON values*

Description

The spread_values() function lets you dive into (potentially nested) JSON objects and extract specific values. spread_values() takes jstring(), jnumber() or jlogical() named function calls as arguments in order to specify the type of the data that should be captured at each desired key location. These values can be of varying types at varying depths.

Usage

```
spread_values(x, ...)
```

Arguments

x	tbl_json object
...	column=value list where 'column' will be the column name created and 'value' must be a call to jstring(), jnumber() or jlogical() specifying the path to get the value (and the type implicit in the function name)

Examples

```
library(magrittr)  # for %>%
'{"name": {"first": "bob", "last": "jones"}, "age": 32}' %>%
  spread_values(
    first.name = jstring("name", "first"),
    age = jnumber("age")
  )
```

tbl_json *Combines structured JSON (as a data.frame) with remaining JSON*

Description

Combines structured JSON (as a data.frame) with remaining JSON

Note that json.list must have the same length as nrow(df), and if json.list has any NULL elements, the corresponding rows will be removed from df. Also note that "..JSON" is a reserved column name used internally for filtering tbl_json objects, and so is not allowed in the data.frame names.

Usage

```
tbl_json(df, json.list, drop.null.json = FALSE)

as.tbl_json(x, ...)

## S3 method for class 'tbl_json'
as.tbl_json(x, ...)

## S3 method for class 'character'
as.tbl_json(x, ...)

## S3 method for class 'data.frame'
as.tbl_json(x, json.column, ...)

is.tbl_json(x)
```

Arguments

df	data.frame
json.list	list of json lists parsed with fromJSON
drop.null.json	drop NULL json entries from data.frame and json
x	an object to convert into a tbl_json object
...	other arguments
json.column	the name of the JSON column of data in x, if x is a data.frame

tidyjson *tidyjson.*

Description

tidyjson.

worldbank *Projects funded by the World Bank*

Description

From: http://jsonstudio.com/resources/

Usage

```
worldbank
```

Format

JSON

Examples

```
library(dplyr)
library(jsonlite)

# Print the first record
worldbank[[1]] %>% prettify

# Get the top 10 sectors by funded project in Africa
wb_sectors <- worldbank %>%   # 500 Projects funded by the world bank
  spread_values(
    name = jstring("project_name"), # Spread name
    region = jstring("regionname")  # Spread region
  ) %>%
  enter_object("majorsector_percent") %>%              # Enter the 'sector' object
  gather_array("sector.index") %>%        # Gather the array
  spread_values(sector = jstring("Name")) # Spread the sector name

# Examine the structured data
wb_sectors %>% head

# Get the top 10 sectors by funded project in Africa
wb_sectors %>%
  filter(region == "Africa") %>% # Filter to just Africa
  group_by(sector) %>%           # Group by sector
  tally() %>%                    # Count funded projects
  arrange(desc(n)) %>%           # Arrange descending
  top_n(10)                      # Take the top 10
```

wrap_dplyr_verb *Wrapper for extending dplyr verbs to tbl_json objects*

Description

Wrapper for extending dplyr verbs to tbl_json objects

Usage

```
wrap_dplyr_verb(dplyr.verb)
```

Arguments

dplyr.verb	a dplyr::verb such as filter, arrange

[.tbl_json *Extract subsets of a tbl_json object (not replace)*

Description

Extends '[.data.frame' to work with tbl_json objects, so that row filtering of the underlying data.frame also filters the associated JSON.

Usage

```
## S3 method for class 'tbl_json'
x[i, j, drop = if (missing(i)) TRUE else length(cols) ==
  1]
```

Arguments

x	a tbl_json object
i	row elements to extract
j	column elements to extract
drop	whether or not to simplify results

Index

∗Topic **datasets**
    allowed_json_types, 2
[.tbl_json, 17

allowed_json_types, 2
append_values, 3
append_values_factory, 3
append_values_logical *(*append_values*)*, 3
append_values_number *(*append_values*)*, 3
append_values_string *(*append_values*)*, 3
append_values_type, 4
as.tbl_json *(*tbl_json*)*, 15

commits, 4
companies, 5

determine_types, 6

enter_object, 6

gather_array, 7
gather_keys, 8

is.tbl_json *(*tbl_json*)*, 15
issues, 9

jfactory, 10
jfunctions, 10
jlogical *(*jfunctions*)*, 10
jnumber *(*jfunctions*)*, 10
json_lengths, 11
json_types, 11
jstring *(*jfunctions*)*, 10

list_path, 12

my_unlist, 12

prep_path, 13

read_json, 13
replace_nulls, 14

spread_values, 14

tbl_json, 15
tidyjson, 15
tidyjson-package *(*tidyjson*)*, 15

worldbank, 16
wrap_dplyr_verb, 17