Package: tidyjson (via r-universe)

September 3, 2024

Title Tidy Complex 'JSON'

Version 0.3.2.9000

Description Turn complex 'JSON' data into tidy data frames.

License MIT + file LICENSE

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

BugReports https://github.com/colearendt/tidyjson/issues

Depends R (>= 3.1.0)

Imports assertthat, dplyr (>= 1.0.0), jsonlite, magrittr, purrr, rlang, tibble, tidyr (>= 1.0.0)

Suggests covr, forcats, ggplot2, igraph, knitr, listviewer, lubridate, RColorBrewer, rmarkdown, rprojroot, testthat (>= 3.0.0), vctrs, viridis, wordcloud

VignetteBuilder knitr

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.2.1

Repository https://colearendt.r-universe.dev

RemoteUrl https://github.com/colearendt/tidyjson

RemoteRef HEAD

RemoteSha 797735b7caf9372e512f9bd33e946d549b9effef

Contents

`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

An object of class `character` of length 6.

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

Description

The append_values functions let you take any scalar JSON values of a given type ("string", "number", "logical") and add them as a new column named column.name. This is particularly useful after using gather_object to gather an object.

Usage

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

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

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

Arguments

.x	a json string or tbl_json object
column.name	the name of the column to append values as
force	should values be coerced to the appropriate type when possible, otherwise, types are checked first (requires more memory)
recursive	logical indicating whether to recurisvely extract a single value from a nested object. Only used when force = TRUE. If force = FALSE, and recursive = TRUE, throws an error.

Details

Any values that can not be converted to the 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 NULL or an object or array.

Note that the append_values functions do not alter the JSON attribute of the tbl_json object in any way.

Value

a tbl_json object

See Also

gather_object to gather an object first, spread_all to spread values into new columns, json_get_column

Examples

```
# Stack names
'{"first": "bob", "last": "jones"}' %>%
  gather_object %>%
  append_values_string

# This is most useful when data is stored in name-value pairs
# For example, tags in recipes:
recipes <- c('{"name": "pie", "tags": {"apple": 10, "pie": 2, "flour": 5}}',
             '{"name": "cookie", "tags": {"chocolate": 2, "cookie": 1}}')
recipes %>%
  spread_values(name = jstring(name)) %>%
  enter_object(tags) %>%
  gather_object("tag") %>%
  append_values_number("count")
```

---

as.character.tbl_json *Convert the JSON in an tbl_json object back to a JSON string*

Description

Convert the JSON in an tbl_json object back to a JSON string

Usage

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

Arguments

x	a tbl_json object
...	not used (map_chr used instead)

Value

a character vector of formatted JSON

as_tibble.tbl_json *Convert a tbl_json back to a tbl_df*

Description

Drops the JSON attribute and the tbl_json class, so that we are back to a pure tbl_df. Useful for some internals. Also useful when you are done processing the JSON portion of your data and are ready to move on to other tools.

Usage

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

as_data_frame.tbl_json(x, ...)
```

Arguments

x	a tbl_json object
...	additional parameters

Details

Note that as.tbl calls tbl_df under the covers, which in turn calls as_tibble. As a result, this should take care of all cases.

Value

a tbl_df object (with no tbl_json component)

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)

# Commits is a long character string
commits %>% nchar

# Let's make it a tbl_json object
commits %>% as.tbl_json

# It begins as an array, so let's gather that
commits %>% gather_array

# Now let's spread all the top level values
commits %>% gather_array %>% spread_all %>% glimpse

# Are there any top level objects or arrays?
commits %>% gather_array %>% gather_object %>% json_types %>%
  count(name, type)

# Let's look at the parents array
commits %>% gather_array("commit") %>%
  enter_object(parents) %>% gather_array("parent") %>%
  spread_all %>% glimpse
```

---

companies *Startup company information for 1,000 companies*

Description

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

Usage

```
companies
```

Format

JSON

Examples

```
library(dplyr)

# Companies is a long character vector
companies %>% str

# Work with a small sample
co_samp <- companies[1:5]
```

```
# Gather top level values and glimpse
co_samp %>% spread_all %>% glimpse

# Get the key employees data for the first 100 companies
key_employees <- companies[1:100] %>%
  spread_all %>%
  select(name) %>%
  enter_object(relationships) %>%
  gather_array() %>%
  spread_all

key_employees %>% glimpse

# Show the top 10 titles
key_employees %>%
  filter(!is_past) %>%
  count(title) %>%
  arrange(desc(n)) %>%
  top_n(10)
```

`enter_object` *Enter into a specific object and discard all other JSON data*

Description

When manipulating a JSON object, `enter_object` lets you navigate to a specific value of the object by referencing it's name. JSON can contain nested objects, and you can pass in more than one character string into `enter_object` to navigate through multiple objects simultaneously.

Usage

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

Arguments

`.x`	a json string or tbl_json object
`...`	a quoted or unquoted sequence of strings designating the object name or sequences of names you wish to enter

Details

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 row will be discarded.

In pipelines, `enter_object` is often preceded by `gather_object` and followed by `gather_array` if the value is an array, or `spread_all` if the value is an object.

Value

a tbl_json object

See Also

gather_object to find sub-objects that could be entered into, gather_array to gather an array in an object and spread_all or spread_values to spread values in an object.

Examples

```
# Let's start with a simple example of parents and children
json <- c('{"parent": "bob",  "children": ["sally", "george"]}',
          '{"parent": "fred", "children": ["billy"]}',
          '{"parent": "anne"}')

# We can see the names and types in each
json %>% gather_object %>% json_types

# Let's capture the parent first and then enter in the children object
json %>% spread_all %>% enter_object(children)

# Also works with quotes
json %>% spread_all %>% enter_object("children")

# Notice that "anne" was discarded, as she has no children

# We can now use gather array to stack the array
json %>% spread_all %>% enter_object(children) %>%
  gather_array("child.num")

# And append_values_string to add the children names
json %>% spread_all %>% enter_object(children) %>%
  gather_array("child.num") %>%
  append_values_string("child")

# The path can be comma delimited to go deep into a nested object
json <- '{"name": "bob", "attributes": {"age": 32, "gender": "male"}}'
json %>% enter_object(attributes, age)

# A more realistc example with companies data
library(dplyr)
companies %>%
  enter_object(acquisitions) %>%
  gather_array %>%
  spread_all %>%
  glimpse
```

gather_array *Gather a JSON array into index-value pairs*

Description

gather_array collapses a JSON array into index-value pairs, creating a new column 'array.index' to store the index of the array, and storing values in the 'JSON' attribute for further tidyjson manipulation. All other columns are duplicated as necessary. This allows you to access the values of the array just like gather_object lets you access the values of an object.

Usage

```
gather_array(.x, column.name = default.column.name)
```

Arguments

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

Details

JSON arrays can be simple vectors (fixed or varying length number, string or logical vectors with or without null values). But they also often contain lists of other objects (like a list of purchases for a user). Thus, the best analogy in R for a JSON array is an unnamed list.

gather_array is often preceded by enter_object when the array is nested under a JSON object, and is often followed by gather_object or enter_object if the array values are objects, or by append_values to append all scalar values as a new column or json_types to determine the types of the array elements (JSON does not guarantee they are the same type).

Value

a tbl_json object

See Also

gather_object to gather a JSON object, enter_object to enter into an object, gather to gather name-value pairs in a data frame

Examples

```
# A simple character array example
json <- '["a", "b", "c"]'

# Check that this is an array
json %>% json_types

# Gather array and check types
json %>% gather_array %>% json_types
```

```
# Extract string values
json %>% gather_array %>% append_values_string

# A more complex mixed type example
json <- '["a", 1, true, null, {"name": "value"}]'

# Then we can use the column.name argument to change the name column
json %>% gather_array %>% json_types

# A nested array
json <- '[["a", "b", "c"], ["a", "d"], ["b", "c"]]'

# Extract both levels
json %>% gather_array("index.1") %>% gather_array("index.2") %>%
  append_values_string

# Some JSON begins as an array
commits %>% gather_array

# We can use spread_all to capture all values
# (recursive = FALSE to limit to the top level object)
library(dplyr)
commits %>% gather_array %>% spread_all(recursive = FALSE) %>% glimpse
```

`gather_object` *Gather a JSON object into name-value pairs*

Description

`gather_object` collapses a JSON object into name-value pairs, creating a new column `'name'` to store the pair names, and storing the values in the `'JSON'` attribute for further tidyjson manipulation. All other columns are duplicated as necessary. This allows you to access the names of the object pairs just like `gather_array` lets you access the values of an array.

Usage

```
gather_object(.x, column.name = default.column.name)
```

Arguments

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

Details

`gather_object` is often followed by `enter_object` to enter into a value that is an object, by `append_values` to append all scalar values as a new column or `json_types` to determine the types of the values.

Value

a tbl_json object

See Also

gather_array to gather a JSON array, enter_object to enter into an object, gather to gather name-value pairs in a data frame

Examples

```
# Let's start with a very simple example
json <- '{"name": "bob", "age": 32, "gender": "male"}'

# Check that this is an object
json %>% json_types

# Gather object and check types
json %>% gather_object %>% json_types

# Sometimes data is stored in object pair names
json <- '{"2014": 32, "2015": 56, "2016": 14}'

# Then we can use the column.name argument to change the column name
json %>% gather_object("year")

# We can also use append_values_number to capture the values, since they are
# all of the same type
json %>% gather_object("year") %>% append_values_number("count")

# This can even work with a more complex, nested example
json <- '{"2015": {"1": 10, "3": 1, "11": 5}, "2016": {"2": 3, "5": 15}}'
json %>% gather_object("year") %>% gather_object("month") %>%
  append_values_number("count")

# Most JSON starts out as an object (or an array of objects), and
# gather_object can be used to inspect the top level (or 2nd level) objects
library(dplyr)
worldbank %>% gather_object %>% json_types %>% count(name, type)
```

---

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)

# issues is a long character string
nchar(issues)

# Let's make it a tbl_json object
issues %>% as.tbl_json

# It begins as an array, so let's gather that
issues %>% gather_array

# Now let's spread all the top level values
issues %>% gather_array %>% spread_all %>% glimpse

# Are there any top level objects or arrays?
issues %>% gather_array %>% gather_object %>% json_types %>%
  count(name, type) %>%
  filter(type %in% c("array", "object"))

# Count issues labels by name
labels <- issues %>%
  gather_array %>%                    # stack issues as "issue.num"
  spread_values(id = jnumber(id)) %>% # capture just issue id
  enter_object(labels) %>%            # filter just those with labels
  gather_array("label.index") %>%     # stack labels
  spread_all
labels %>% glimpse

# Count number of distinct issues each label appears in
labels %>%
  group_by(name) %>%
  summarize(num.issues = n_distinct(id))
```

`is_json` *Predicates to test for specific JSON types in* `tbl_json` *objects*

Description

These functions are often useful with `filter` to filter complex JSON by type before applying `gather_object` or `gather_array`.

Usage

```
is_json_string(.x)

is_json_number(.x)

is_json_logical(.x)

is_json_null(.x)

is_json_array(.x)

is_json_object(.x)

is_json_scalar(.x)
```

Arguments

.x	a json string or tbl_json object

Value

a logical vector

See Also

json_types for creating a new column to identify the type of every JSON document

Examples

```
# Test a simple example
json <- '[1, "string", true, [1, 2], {"name": "value"}, null]' %>% gather_array
json %>% is_json_number
json %>% is_json_array
json %>% is_json_scalar

# Use with filter
library(dplyr)
json %>% filter(is_json_object(.))

# Combine with filter in advance of using gather_array
companies[1:5] %>% gather_object %>% filter(is_json_array(.))
companies[1:5] %>% gather_object %>% filter(is_json_array(.)) %>%
  gather_array

# Combine with filter in advance of using gather_object
companies[1:5] %>% gather_object %>% filter(is_json_object(.))
companies[1:5] %>% gather_object %>% filter(is_json_object(.)) %>%
  gather_object("name2")
```

`json_complexity` *Compute the complexity (recursively unlisted length) of JSON data*

Description

When investigating complex JSON data it can be helpful to identify the complexity of deeply nested documents. The `json_complexity` function adds a column (default name `"complexity"`) that contains the 'complexity' of the JSON associated with each row. Essentially, every on-null scalar value is found in the object by recursively stripping away all objects or arrays, and the complexity is the count of these scalar values. Note that 'null' has complexity 0, as do empty objects and arrays.

Usage

```
json_complexity(.x, column.name = "complexity")
```

Arguments

`.x`	a json string or tbl_json object
`column.name`	the name to specify for the length column

Value

a `tbl_json` object

See Also

`json_lengths` to compute the length of each value

Examples

```
# A simple example
json <- c('[1, 2, [3, 4]]', '{"k1": 1, "k2": [2, [3, 4]]}', '1', 'null')

# Complexity is larger than length for nested objects
json %>% json_lengths %>% json_complexity

# Worldbank has complexity ranging from 8 to 17
library(magrittr)
worldbank %>% json_complexity %$% table(complexity)

# Commits are much more regular
commits %>% gather_array %>% json_complexity %$% table(complexity)
```

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

Description

Note that these functions fail if they encounter the incorrect type. Note that `jnumber()` is an alias for `jdouble()`.

Usage

```
jstring(..., recursive = FALSE)

jlogical(..., recursive = FALSE)

jinteger(..., recursive = FALSE)

jdouble(..., recursive = FALSE)

jnumber(..., recursive = FALSE)
```

Arguments

`...`	a quoted or unquoted sequence of strings designating the object name sequence you wish to follow to find a value
`recursive`	logical indicating whether second level and beyond objects should be extracted. Only works when there exists a single value in the nested json object

Value

a function that can operate on parsed JSON data

See Also

spread_values for using these functions to spread the values of a JSON object into new columns

json_get *Get JSON from a tbl_json*

Description

Extract the raw JSON from a tbl_json object. This is equivalent to reading the "..JSON" hidden column. But is a helper in case of future behavior changes. This replaces previous behavior, where the raw JSON was stored in an attribute.

Usage

```
json_get(.data)
```

Arguments

.data	A tbl_json object

Value

A nested list representing the JSON data

---

json_get_column *Make the JSON data a persistent column*

Description

Extract the raw JSON from a tbl_json object. Store it in a column. WARNING: column name collisions will be overwritten

Usage

```
json_get_column(.data, column_name = "json")
```

Arguments

.data	A tbl_json object
column_name	Optional. The name of the output column (either as a string or unquoted name). Default "json"

Value

A tbl_json object with an added column containing the JSON data

Examples

```
tj <- as_tbl_json('{"a": "b"}')
json_get_column(tj, my_json)
```

`json_lengths` *Compute the length of 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 name-value pairs. For arrays, this will be equal to the length of the array. All scalar values will be of length 1, and null will have length 0.

Usage

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

Arguments

`.x`	a json string or `tbl_json` object
`column.name`	the name to specify for the length column

Value

a `tbl_json` object

See Also

`json_complexity` to compute the recursive length of each value

Examples

```
# A simple example
json <- c('[1, 2, 3]', '{"k1": 1, "k2": 2}', '1', 'null')

# Complexity is larger than length for nested objects
json %>% json_lengths

# Worldbank objcts are either length 7 or 8
library(magrittr)
worldbank %>% json_lengths %$% table(length)

# All commits are length 8
commits %>% gather_array %>% json_lengths %$% table(length)
```

`json_schema` *Create a schema for a JSON document or collection*

Description

Returns a JSON document that captures the 'schema' of the collection of document(s) passed in, as a JSON string. The schema collapses complex JSON into a simple form using the following rules:

Usage

```
json_schema(.x, type = c("string", "value"))
```

Arguments

`.x`	a json string or `tbl_json` object
`type`	whether to capture scalar nodes using the string that defines their type (e.g., "logical") or as a representative value (e.g., "true")

Details

- string -> "string", e.g., "a sentence" -> "string"
- number -> "number", e.g., 32000.1 -> "number"
- true -> "logical", e.g., true -> "logical"
- false -> "logical", e.g., false -> "logical"
- null -> "null", e.g., null -> "null"
- array -> [<type>] e.g., [1, 2] -> ["number"]
- object -> "name": <type> e.g., "age": 32 -> "age": "number"

For more complex JSON objects, ties are broken by taking the most complex example (using `json_complexity`), and then by type (using `json_types`).

This means that if a name has varying schema across documents, the most complex schema will be chosen as being representative. Similarly, if the elements of an array vary in schema, the most complex element is chosen, and if arrays vary in schema across documents, the most complex is chosen.

Note that `json_schema` can be slow for large JSON document collections, you may want to sample your JSON collection first.

Value

a character string JSON document that represents the schema of the collection

See Also

`json_structure` to recursively structure all documents into a single data frame

Examples

```
# A simple string
'"string"' %>% json_schema %>% writeLines

# A simple object
'{"name": "value"}' %>% json_schema %>% writeLines

# A more complex JSON array
json <- '[{"a": 1}, [1, 2], "a", 1, true, null]'

# Using type = 'string' (default)
json %>% json_schema %>% writeLines

# Using type = 'value' to show a representative value
json %>% json_schema(type = "value") %>% writeLines

# Schema of the first 5 github issues
## Not run:
  library(dplyr)
  issues %>% gather_array %>% slice(1:10) %>%
    json_schema(type = "value") %>% writeLines

## End(Not run)
```

`json_structure` *Recursively structures arbitrary JSON data into a single data frame*

Description

Returns a `tbl_json` object where each row corresponds to a leaf in the JSON structure. The first row corresponds to the JSON document as a whole. If the document is a scalar value (JSON string, number, logical or null), then there will only be 1 row. If instead it is an object or an array, then subsequent rows will recursively correspond to the elements (and their children) of the object or array.

Usage

```
json_structure(.x)
```

Arguments

`.x`	a json string or tbl_json object

Details

The columns in the `tbl_json` returend are defined as

- `document.id` 1L if `.x` is a single JSON string, otherwise the index of `.x`.
- `parent.id` the string identifier of the parent node for this child.

- `level` what level of the hierarchy this child resides at, starting at `0L` for the root and incrementing for each level of nested array or object.
- `index` what index of the parent object / array this child resides at (from `gather_array` for arrays).
- `child.id` a unique ID for this leaf in this document, represented as <parent>.<index> where <parent> is the ID for the parent and <index> is this index.
- `seq` the sequence of names / indices that led to this child (parents that are arrays are excluded) as a list, where character strings denote objects and integers denote array positions
- `name` if this is the value of an object, what was the name that it is listed under (from `gather_object`).
- `type` the type of this object (from `json_types`).
- `length` the length of this object (from `json_lengths`).

Value

a `tbl_json` object

See Also

`json_schema` to create a schema for a JSON document or collection

Examples

```
# A simple string
'"string"' %>% json_structure

# A simple object
'{"name": "value"}' %>% json_structure

# A complex array
'[{"a": 1}, [1, 2], "a", 1, true, null]' %>% json_structure

# A sample of structure rows from a company
library(dplyr)
companies[1] %>% json_structure %>% sample_n(5)
```

---

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

Description

The function `json_types` inspects the JSON associated with each row of the `tbl_json` object, 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 json string or 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 often follows after gather_array, gather_object or enter_object to inspect the types of the elements of JSON objects or arrays.

Value

a tbl_json object

Examples

```
# A simple example
c('{"a": 1}', '[1, 2]', '"a"', '1', 'true', 'null') %>% json_types

# Type distribution in the first 10 companies
library(dplyr)
companies[1:10] %>% gather_object %>% json_types %>% count(type)
```

---

print.tbl_json *Print a tbl_json object*

Description

Print a tbl_json object

Usage

```
## S3 method for class 'tbl_json'
print(x, ..., json.n = 20, json.width = 15)
```

Arguments

x	a tbl_json object
...	other arguments into print.tbl_df
json.n	number of json records to add (...) otherwise
json.width	number of json characters to print

rbind_tbl_json *Bind two tbl_json objects together and preserve JSON attribute*

Description

Bind two tbl_json objects together and preserve JSON attribute

Usage

```
rbind_tbl_json(x, y)
```

Arguments

x	a tbl_json object
y	a tbl_json_object

Value

x and y row-binded together with appropriate JSON attribute

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

Description

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

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

a tbl_json object

spread_all *Spreads all scalar values of a JSON object into new columns*

Description

Like the spread function in tidyr but for JSON, this function spreads out any JSON objects that are scalars into new columns. If objects are nested, then the recursive flag will expand scalar values of nested objects out with a compound column name based on the sequences of nested object names concatenated with the sep character.

Usage

```
spread_all(.x, recursive = TRUE, sep = ".")
```

Arguments

.x	a json string or tbl_json object
recursive	whether or not to recursively spread nested objects
sep	character used to separate nested object names when resursive is TRUE

Details

Note that arrays are ignored by this function, use gather_array to gather the array first, and then use spread_all if the array contains objects or use one of the append_values functions to capture the array values if they are scalars.

Note that scalar JSON values (e.g., a JSON string like '1') are also ignored, as they have no names to create column names with.

The order of columns is determined by the order they are encountered in the JSON document, with nested objects placed at the end.

If an objects have name-value pairs with names that are duplicates, then ".n" is appended for n incrementing from 2 on to ensure that columns are unique. This also happens if .x already has a column with the name of a name-value pair.

This function does not change the value of the JSON attribute of the tbl_json object in any way.

Value

a tbl_json object

See Also

spread_values to specific which specific values to spread along with their types, spread for spreading data frames

Examples

```
# A simple example
json <- c('{"a": "x", "b": 1, "c": true}',
          '{"a": "y", "c": false}',
          '{"a": null, "d": "z"}')
json %>% spread_all

# A more complex example
worldbank %>% spread_all

## Not run:
  # Resolving duplicate column names
  json <- '{"a": "x", "a": "y"}'
  json %>% spread_all

## End(Not run)
```

`spread_values` *Spreads specific scalar values of a JSON object into new columns*

Description

The `spread_values` function lets you extract extract specific values from (potentiall nested) JSON objects. `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 name-value pair location. These values can be of varying types at varying depths.

Usage

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

Arguments

`.x`	a json string or `tbl_json` object
`...`	`column = value` pairs 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)

Details

Note that `jstring`, `jnumber` and `jlogical` will fail if they encounter the incorrect type in any document.

The advantage of `spread_values` over `spread_all` is that you are guaranteed to get a consistent data frame structure (columns and types) out of any `spread_values` call. `spread_all` requires less typing, but because it infers the columns and their types from the JSON, it is less suitable when programming.

Value

a tbl_json object

See Also

spread_all for spreading all values, spread for spreading data frames, jstring, jnumber, jlogical for accessing specific names

Examples

```
# A simple example
json <- '{"name": {"first": "Bob", "last": "Jones"}, "age": 32}'

# Using spread_values
json %>%
  spread_values(
    first.name = jstring(name, first),
    last.name  = jstring(name, last),
    age        = jnumber(age)
  )

# Another document, this time with a middle name (and no age)
json2 <- '{"name": {"first": "Ann", "middle": "A", "last": "Smith"}}'

# spread_values still gives the same column structure
c(json, json2) %>%
  spread_values(
    first.name = jstring(name, first),
    last.name  = jstring(name, last),
    age        = jnumber(age)
  )

# whereas spread_all adds a new column
json %>% spread_all
c(json, json2) %>% spread_all
```

---

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

Description

Constructs a tbl_json object, for further downstream manipulation by other tidyjson functions. Methods exist to convert JSON stored in character strings without any other associated data, as a separate character string and associated data frame, or as a single data frame with a specified character string JSON column.

Usage

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

as.tbl_json(.x, ...)

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 'list'
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 df and json.list
...	other arguments
.column_order	Experimental argument to preserve column order for the hidden column
.x	an object to convert into a tbl_json object
json.column	the name of the json column of data in .x, if .x is a data frame

Details

Most tidyjson functions accept a tbl_json object as the first argument, and return a tbl_json object unless otherwise specified. tidyjson functions will attempt to convert an object that isn't a tbl_json object first, and so explicit construction of tidyjson objects is rarely needed.

tbl_json objects consist of a data frame along with it's associated JSON, where each row of the data frame corresponds to a single JSON document. The JSON is stored in a "JSON" attribute.

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 names of df.

Value

a tbl_json object

See Also

read_json for reading json from files

Examples

```
# Construct a tbl_json object using a charater string of JSON
json <- '{"animal": "cat", "count": 2}'
json %>% as.tbl_json

# access the "JSON" argument
json %>% as.tbl_json %>% attr("JSON")

# Construct a tbl_json object using multiple documents
json <- c('{"animal": "cat", "count": 2}', '{"animal": "parrot", "count": 1}')
json %>% as.tbl_json

# Construct a tbl_json object from a data.frame with a JSON colum
library(tibble)
farms <- tribble(
  ~farm, ~animals,
  1L,    '[{"animal": "pig", "count": 50}, {"animal": "cow", "count": 10}]',
  2L,    '[{"animal": "chicken", "count": 20}]'
)
farms %>% as.tbl_json(json.column = "animals")
# tidy the farms
farms %>% as.tbl_json(json.column = "animals") %>%
  gather_array %>% spread_all
```

---

tidyjson *tidyjson*

Description

The tidyjson package provides tools to turn complex JSON data into tidy tibbles and data frames.

---

worldbank *Projects funded by the World Bank*

Description

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

Usage

```
worldbank
```

Format

JSON

Examples

```
## Not run:
library(dplyr)

# worldbank is a 500 length character vector of JSON
length(worldbank)

# Let's look at top level values
worldbank %>% spread_all %>% glimpse

# Are there any arrays?
worldbank %>% gather_object %>% json_types %>% count(name, type)

# Get the top 10 sectors by funded project in Africa
wb_sectors <- worldbank %>%   # 500 Projects funded by the world bank
  spread_all %>%
  select(project_name, regionname) %>%
  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 %>% glimpse

# Get the top 10 sectors by funded project in Africa
wb_sectors %>%
  filter(regionname == "Africa") %>% # Filter to just Africa
  count(sector) %>%              # Count by sector
  arrange(desc(n)) %>%           # Arrange descending
  top_n(10)                      # Take the top 10

## End(Not run)
```

`[.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 = FALSE]
```

Arguments

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

Value

a tbl_json object

Index