Package ‘tableschema.r’

November 14, 2018

Type  Package
Title  Frictionless Data Table Schema
Version  1.1.0
Date  2018-11-5
Maintainer  Kleanthis Koupidis <koupidis@okfn.gr>

Description  Allows to work with 'Table Schema' (<http://specs.frictionlessdata.io/table-schema/>). 'Table Schema' is well suited for use cases around handling and validating tabular data in text formats such as 'csv', but its utility extends well beyond this core usage, towards a range of applications where data benefits from a portable schema format. The 'tableschema.r' package can load and validate any table schema descriptor, allow the creation and modification of descriptors, expose methods for reading and streaming data that conforms to a 'Table Schema' via the 'Tabular Data Resource' abstraction.

URL  https://github.com/frictionlessdata/tableschema-r

BugReports  https://github.com/frictionlessdata/tableschema-r/issues

License  MIT + file LICENSE

Encoding  UTF-8

LazyData  true

Imports  config, future, httr, iterators, jsonlite, jsonvalidate, lubridate, purrr, R6, RCurl, rlist, stringr, urltools

Suggests  covr, foreach, testthat

Collate  'constraints.checkUnique.R' 'constraints.checkRequired.R'
           'constraints.checkPattern.R' 'constraints.checkMinLength.R'
           'constraints.checkMinimum.R' 'constraints.checkMaxLength.R'
           'constraints.checkMaxValue.R' 'constraints.checkEnum.R'
           'constraints.R' 'types.castArray.R' 'types.castYearmonth.R'
           'types.castYear.R' 'types.castTime.R' 'types.castString.R'
           'types.castObject.R' 'types.castNumber.R' 'types.castList.R'
           'types.castInteger.R' 'types.castGeopoint.R'
           'types.castGeojson.R' 'types.castDuration.R'
           'types.castDatetime.R' 'types.castDate.R' 'types.castBoolean.R'
           'types.castAny.R' 'types.R' 'field.R' 'helpers.R' 'infer.R'
R topics documented:

'is.valid.R' 'tableschemaerror.R' 'profile.R' 'readable.R'
'readable.array.R' 'readable.connection.R' 'schema.R' 'table.R'
'tableschema.r.R' 'validate.R' 'writable.R'

RoxygenNote 6.1.0.9000

NeedsCompilation no

Author Kleanthis Koupidis [aut, cre],
Lazaros Ioannidis [aut],
Charalampos Bratsas [aut],
Open Knowledge International [cph]

Repository CRAN

Date/Publication 2018-11-14 16:50:03 UTC

R topics documented:

tableschema.r-package ........................................ 3
Constraints ...................................................... 7
constraints.checkEnum ......................................... 7
constraints.checkMaximum ....................................... 8
constraints.checkMaxLength ..................................... 9
constraints.checkMinimum ...................................... 10
constraints.checkMinLength ................................... 10
constraints.checkPattern ....................................... 11
constraints.checkRequired ..................................... 12
constraints.checkUnique ....................................... 13
DEFAULT_DECIMAL_CHAR ....................................... 13
DEFAULT_GROUP_CHAR .......................................... 14
durations ......................................................... 14
FALSE_VALUES .................................................. 15
Field .............................................................. 15
helpers.expandFieldDescriptor ................................. 18
helpers.expandSchemaDescriptor .............................. 18
helpers.from.json.to.list ..................................... 19
helpers.from.list.to.json ...................................... 19
helpers.retrieveDescriptor .................................... 19
infer ............................................................... 20
is.binary ......................................................... 21
is.email .......................................................... 21
is.uri ............................................................ 22
is.uuid .......................................................... 22
is.valid ......................................................... 23
is_empty .......................................................... 23
is_integer ......................................................... 24
is_object ......................................................... 24
Profile .......................................................... 24
Profile.load ...................................................... 25
Readable .......................................................... 26
Description

Table class for working with data and schema

Introduction

Table Schema is a simple language- and implementation-agnostic way to declare a schema for tabular data. Table Schema is well suited for use cases around handling and validating tabular data in text formats such as CSV, but its utility extends well beyond this core usage, towards a range of applications where data benefits from a portable schema format.

Concepts

#
Tabular data

Tabular data consists of a set of rows. Each row has a set of fields (columns). We usually expect that each row has the same set of fields and thus we can talk about the fields for the table as a whole.

In case of tables in spreadsheets or CSV files we often interpret the first row as a header row, giving the names of the fields. By contrast, in other situations, e.g. tables in SQL databases, the field names are explicitly designated.

Physical and logical representation

In order to talk about the representation and processing of tabular data from text-based sources, it is useful to introduce the concepts of the physical and the logical representation of data.

The physical representation of data refers to the representation of data as text on disk, for example, in a CSV or JSON file. This representation may have some type information (JSON, where the primitive types that JSON supports can be used) or not (CSV, where all data is represented in string form).

The logical representation of data refers to the "ideal" representation of the data in terms of primitive types, data structures, and relations, all as defined by the specification. We could say that the specification is about the logical representation of data, as well as about ways in which to handle conversion of a physical representation to a logical one.

In this document, we’ll explicitly refer to either the physical or logical representation in places where it prevents ambiguity for those engaging with the specification, especially implementors.

For example, constraints should be tested on the logical representation of data, whereas a property like missingValues applies to the physical representation of the data.

Descriptor

A Table Schema is represented by a descriptor. The descriptor MUST be a JSON object (JSON is defined in RFC 4627).

It MUST contain a property fields. fields MUST be an array/list where each entry in the array/list is a field descriptor (as defined below). The order of elements in fields array/list MUST be the order of fields in the CSV file. The number of elements in fields array/list SHOULD be exactly the same as the number of fields in the CSV file.

The descriptor MAY have the additional properties set out below and MAY contain any number of other properties (not defined in this specification).

Field Descriptors

See Field Class

Types and Formats

See Types Class

Constraints

See Constraints Class
Other Properties

In addition to field descriptors, there are the following "table level" properties.

Missing Values

Many datasets arrive with missing data values, either because a value was not collected or it never existed. Missing values may be indicated simply by the value being empty in other cases a special value may have been used e.g. -, NaN, 0, -9999 etc.

missingValues dictates which string values should be treated as null values. This conversion to null is done before any other attempted type-specific string conversion. The default value list("") means that empty strings will be converted to null before any other processing takes place. Providing the empty list means that no conversion to null will be done, on any value.

missingValues MUST be a list where each entry is a string.

Why strings: missingValues are strings rather than being the data type of the particular field. This allows for comparison prior to casting and for fields to have missing value which are not of their type, for example a number field to have missing values indicated by -. Examples:

- missingValues = list(""")
- missingValues = list("-")
- missingValues = list("NaN", "-")

Primary Key

A primary key is a field or set of fields that uniquely identifies each row in the table.

The primaryKey entry in the schema object is optional. If present it specifies the primary key for this table.

The primaryKey, if present, MUST be:

- Either: an array of strings with each string corresponding to one of the field name values in the fields array (denoting that the primary key is made up of those fields). It is acceptable to have an array with a single value (indicating just one field in the primary key). Strictly, order of values in the array does not matter. However, it is RECOMMENDED that one follow the order the fields in the fields has as client applications may utitlize the order of the primary key list (e.g. in concatenating values together).
- Or: a single string corresponding to one of the field name values in the fields array/list (indicating that this field is the primary key). Note that this version corresponds to the array form with a single value (and can be seen as simply a more convenient way of specifying a single field primary key).

Foreign Keys

A foreign key is a reference where values in a field (or fields) on the table ('resource' in data package terminology) described by this Table Schema connect to values a field (or fields) on this or a separate table (resource). They are directly modelled on the concept of foreign keys in SQL.

The foreignKeys property, if present, MUST be a list Each entry in the array must be a foreignKey. A foreignKey MUST be an object and MUST have the following properties:
• **fields** - fields is a string or array specifying the field or fields on this resource that form the source part of the foreign key. The structure of the string or array is as per `primaryKey` above.

• **reference** - reference MUST be a object. The object
  
  – MUST have a property `resource` which is the name of the resource within the current data package (i.e. the data package within which this Table Schema is located). For self-referencing foreign keys, i.e. references between fields in this Table Schema, the value of `resource` MUST be "" (i.e. the empty string).
  
  – MUST have a property `fields` which is a string if the outer `fields` is a string, else an array of the same length as the outer `fields`, describing the field (or fields) references on the destination resource. The structure of the string or array is as per `primaryKey` above.

**Comment**: Foreign Keys create links between one Table Schema and another Table Schema, and implicitly between the data tables described by those Table Schemas. If the foreign key is referring to another Table Schema how is that other Table Schema discovered? The answer is that a Table Schema will usually be embedded inside some larger descriptor for a dataset, in particular as the schema for a resource in the resources array of a hrefhttp://frictionlessdata.io/specs/data-package/Data Package. It is the use of Table Schema in this way that permits a meaningful use of a non-empty `resource` property on the foreign key.

**Details**

**jsonlite package** is internally used to convert json data to list objects. The input parameters of functions could be json strings, files or lists and the outputs are in list format to easily further process your data in R environment and exported as desired. More details about handling json you can see jsonlite documentation or vignettes here.

**Future package** is also used to load and create Table and Schema classes asynchronously. To retrieve the actual result of the loaded Table or Schema you have to use `value` function to the variable you stored the loaded Table/Schema. More details about future package and sequential and parallel processing you can find here.

Examples section of each function show how to use jsonlite and future packages with tableschema.r. Term array refers to json arrays which if converted in R will be list objects.

**Language**

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this package documents are to be interpreted as described in RFC 2119.

**See Also**

Table Schema Specifications
### Constraints

**Constraints class**

#### Description

R6 class with constraints.

The constraints property on Table Schema Fields can be used by consumers to list constraints for validating field values. For example, validating the data in a Tabular Data Resource against its Table Schema; or as a means to validate data being collected or updated via a data entry interface.

All constraints MUST be tested against the logical representation of data, and the physical representation of constraint values MAY be primitive types as possible in JSON, or represented as strings that are castable with the type and format rules of the field.

#### Usage

Constraints

#### Format

R6Class object.

#### Value

Object of R6Class.

#### See Also

Constraints specifications, constraints.checkEnum, constraints.checkMaximum, constraints.checkMaxLength, constraints.checkMinimum, constraints.checkMinLength, constraints.checkPattern, constraints.checkRequired, constraints.checkUnique

---

### constraints.checkEnum

**Check Enum**

#### Description

Check if the value is exactly match a constraint.

#### Usage

constraints.checkEnum(constraint, value)

#### Arguments

- constraint: numeric list, matrix or vector with the constraint values
- value: numeric value to meet the constraint
Value

TRUE if value meets the constraint

See Also

Constraints specifications

Examples

```
constraints.checkEnum(constraint = list(1, 2), value = 1)
constraints.checkEnum(constraint = list(1, 2), value = 3)
```

Description

Specifies a maximum value for a field. This is different to `maxLength` which checks the number of items in the value. A maximum value constraint checks whether a field value is equal to or less than the specified value. The range checking depends on the type of the field. E.g. an integer field may have a maximum value of 100. If a maximum value constraint is specified then the field descriptor MUST contain a type key.

Usage

```
constraints.checkMaximum(constraint, value)
```

Arguments

- `constraint`: numeric constraint value
- `value`: numeric value to meet the constraint

Value

TRUE if value is equal to or less than the constraint

See Also

Constraints specifications
**Examples**

```python
constraints.checkMaximum(constraint = list(2), value = 1)

constraints.checkMaximum(constraint = 2, value = 3)
```

---

**Description**

Specify the maximum length of a character

**Usage**

```python
constraints.checkMaxLength(constraint, value)
```

**Arguments**

- `constraint`  
  numeric constraint, maximum character length  
- `value`  
  character to meet the constraint

**Value**

TRUE if character length is equal to or less than the constraint

**See Also**

- Constraints specifications

**Examples**

```python
constraints.checkMaxLength(constraint = list(2), value = "hi")

constraints.checkMaxLength(constraint = 2, value = "hello")
```
constraints.checkMinimum

*Check if minimum constraint is met*

**Description**

Specifies a minimum value for a field. This is different to minLength which checks the number of items in the value. A minimum value constraint checks whether a field value is greater than or equal to the specified value. The range checking depends on the type of the field. E.g. an integer field may have a minimum value of 100. If a minimum value constraint is specified then the field descriptor MUST contain a type key.

**Usage**

```plaintext
constraints.checkMinimum(constraint, value)
```

**Arguments**

- **constraint**
  - numeric constraint value
- **value**
  - numeric value to meet the constraint

**Value**

TRUE if value is equal to or greater than the constraint

**See Also**

Constraints specifications

**Examples**

```plaintext
constraints.checkMinimum(constraint = list(2), value = 1)

constraints.checkMinimum(constraint = 2, value = 3)
```

---

constraints.checkMinLength

*Check if minimum character length constraint is met*

**Description**

Specify the minimum length of a character
Usage

constraints.checkMinLength(constraint, value)

Arguments

constraint numeric constraint, minimum character length
value character to meet the constraint

Value

TRUE if character length is equal to or greater than the constraint

See Also

Constraints specifications

Examples

constraints.checkMinLength(constraint = list(3), value = "hi")

constraints.checkMinLength(constraint = 2, value = "hello")

Constraints.checkPattern

Pattern matching

Description

Search for pattern matches (value) within a character vector (constraint). A regular expression is used to test field values. If the regular expression matches then the value is valid. The values of this field MUST conform to the standard XML Schema regular expression syntax.

Usage

constraints.checkPattern(constraint, value)

Arguments

constraint character vector where matches are sought
value character string to be matched

Value

TRUE if the pattern constraint is met
constraints.checkRequired

See Also

Constraints specifications

Examples

    constraints.checkPattern(constraint = '^test$', value = 'test')
    constraints.checkPattern(constraint = '^test$', value = 'TEST')

    constraints.checkRequired(constraint = TRUE, value = 1)
    constraints.checkRequired(constraint = TRUE, value = 0)
    constraints.checkRequired(constraint = TRUE, value = NULL)
    constraints.checkRequired(constraint = TRUE, value = "undefined")
**constraints.checkUnique**

*Check if a field is unique*

**Description**

If TRUE, then all values for that field MUST be unique within the data file in which it is found.

**Usage**

```r
constraints.checkUnique(constraint, value)
```

**Arguments**

- `constraint` set TRUE to check unique values
- `value` value to check

**Value**

TRUE if field is unique

**See Also**

*Constraints specifications*

**Examples**

```r
constraints.checkUnique(constraint = FALSE, value = "any")
constraints.checkUnique(constraint = TRUE, value = "any")
```

---

**DEFAULT_DECIMAL_CHAR**

*default decimal char*

**Description**

default decimal char

**Usage**

```
DEFAULT_DECIMAL_CHAR
```

**Format**

An object of class character of length 1.
**default group char**

**Description**

default group char

**Usage**

```r
DEFAULT_GROUP_CHAR
```

**Format**

An object of class character of length 1.

---

**durations**

**Durations**

**Description**

Help function to use with `types.castDuration`

**Usage**

```r
durations(years = 0, months = 0, days = 0, hours = 0, minutes = 0, seconds = 0)
```

**Arguments**

- `years`
- `months`
- `days`
- `hours`
- `minutes`
- `seconds`

**See Also**

`types.castDuration`
FALSE_VALUES

default false values

Description

default false values

Usage

FALSE_VALUES

Format

An object of class character of length 4.

Field

Field class

Description

Class represents field in the schema.

Data values can be cast to native R types. Casting a value will check the value is of the expected
type, is in the correct format, and complies with any constraints imposed by a schema.

Usage

# Field$new(descriptor, missingValues = list(""))

Arguments

descriptor Schema field descriptor
missingValues A list with vector strings representing missing values

Format

R6Class object.
Details

A field descriptor MUST be a JSON object that describes a single field. The descriptor provides additional human-readable documentation for a field, as well as additional information that may be used to validate the field or create a user interface for data entry.

The field descriptor object MAY contain any number of other properties. Some specific properties are defined below. Of these, only the name property is REQUIRED.

name The field descriptor MUST contain a name property. This property SHOULD correspond to the name of field/column in the data file (if it has a name). As such it SHOULD be unique (though it is possible, but very bad practice, for the data file to have multiple columns with the same name). name SHOULD NOT be considered case sensitive in determining uniqueness. However, since it should correspond to the name of the field in the data file it may be important to preserve case.

title A human readable label or title for the field.
description A description for this field e.g. "The recipient of the funds".

Value

Object of R6Class.

Methods

Field$new(descriptor, missingValues = list("")) Constructor to instantiate Field class.

- descriptor Schema field descriptor.
- missingValues A list with vector strings representing missing values.
- TableSchemaError Raises any error occurred in the process.
- Field Returns Field class instance.

cast_value(value, constraints=TRUE) Cast given value according to the field type and format.

- value Value to cast against field
- constraints Gets constraints configuration: it could be set to true to disable constraint checks, or it could be a List of constraints to check
- errors$TableSchemaError Raises any error occurred in the process
- any Returns cast value

testValue(value, constraints=TRUE) Test if value is compliant to the field.

- value Value to cast against field
- constraints Constraints configuration
- boolean Returns if value is compliant to the field

Properties

name Returns field name

type Returns field type

format Returns field format
Field

required Returns TRUE if field is required
constraints Returns list with field constraints
descriptor Returns field descriptor

Language

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this package documents are to be interpreted as described in RFC 2119.

See Also

Field Descriptors Specifications

Examples

```
DESCRIPTOR = list(name = "height", type = "number")
field <- Field$new(descriptor = DESCRIPTOR)

# get correct instance
fieldName
field$format
field$type

# return true on test
testValue(1)

# cast value
cast_value(1)

# expand descriptor by defaults
field <- Field$new(descriptor = list(name = "name"))
field$descriptor

# parse descriptor with "enum" constraint
field <- Field$new(descriptor = list(name = "status", type = "string",
  constraints = list(enum = list('active', 'inactive'))))
field$testValue('active')
field$testValue('inactive')
field$testValue('activia')
cast_value('active')

# parse descriptor with "minimum" constraint
field <- Field$new(descriptor = list(name = "length", type = "integer",
  constraints = list(minimum = 100)))
field$testValue(200)
field$testValue(50)
```
helpers.expandFieldDescriptor

*Expand Field Descriptor*

**Description**

Helper function to expand field descriptor

**Usage**

`helpers.expandFieldDescriptor(descriptor)`

**Arguments**

`descriptor` descriptor

helpers.expandSchemaDescriptor

*Expand Schema Descriptor*

**Description**

Helper function to expand schema descriptor

**Usage**

`helpers.expandSchemaDescriptor(descriptor)`

**Arguments**

`descriptor` descriptor

```r
# parse descriptor with "maximum" constraint'
field <- Field$new(descriptor = list(name = "length", type = "integer",
                           constraints = list(maximum = 100)))

field$testValue(50)
field$testValue(200)
```
**helpers.from.json.to.list**

*Convert json to list*

**Description**
Convert json to list

**Usage**
helpers.from.json.to.list(lst)

**Arguments**
- lst : list

---

**helpers.from.json.to.list**

*Convert json to list*

**Description**
Convert list to json

**Usage**
helpers.from.list.to.json(json)

**Arguments**
- json : json string

---

**helpers.retrieveDescriptor**

*Retrieve Descriptor*

**Description**
Helper function to retrieve descriptor

**Usage**
helpers.retrieveDescriptor(descriptor)

**Arguments**
- descriptor : descriptor
infer

Infer source schema

Description

Given data source and headers infer will return a Table Schema based on the data values.

Usage

infer(source, options = list())

Arguments

- **source**: data source, one of:
  - string with the local CSV file (path)
  - string with the remote CSV file (url)
  - list of lists representing the rows
  - readable stream with CSV file contents
  - function returning readable stream with CSV file contents

- **options**: any `Table.load` options

Value

Schema descriptor

Examples

```r
# list of lists data source
source = list(
  list("id" = 1,
       "age" = 39,
       "name" = "Paul"),
  list("id" = 2,
       "age" = 23,
       "name" = "Jimmy"),
  list("id" = 3,
       "age" = 36,
       "name" = "Jane"),
  list("id" = 4,
       "age" = 28,
       "name" = "Judy"))

infer(source, options=list(headers=list("id","age","name")))$fields
```
is.binary  Is binary

Description
Is binary

Usage
is.binary(x)

Arguments
x  input value to check

Value
TRUE if binary

is.email  Is email

Description
Is email

Usage
is.email(x)

Arguments
x  email string

Value
TRUE if x is email
is.uri  
Is uri

Description
Is uri

Usage
is.uri(uri)

Arguments
uri  uri input

Value
TRUE if uri string

is.uuid  
Is uuid

Description
Is uuid

Usage
is.uuid(x)

Arguments
x  character

Value
TRUE if uuid
is.valid

Description
Validate a descriptor over a schema

Usage
is.valid(descriptor, schema = NULL)

Arguments
- descriptor: descriptor, one of:
  - string with the local CSV file (path)
  - string with the remote CSV file (url)
  - list object
- schema: Contents of the json schema, or a filename containing a schema

Value
TRUE if valid

is_empty

Description
Is empty list

Usage
is_empty(x)

Arguments
- x: list object
### is_integer

**Description**
Is integer

**Usage**
```
is_integer(x)
```

**Arguments**

- **x**: number

### is_object

**Description**
Is object

**Usage**
```
is_object(x)
```

**Arguments**

- **x**: list, array, json string

### Profile

**Description**
Class to represent JSON Schema profile from Profiles Registry.

**Usage**
```
# Profile.load(profile)
```

**Arguments**

- **profile**: string profile name in registry or URL to JSON Schema
Profile.load

Format

R6Class object.

Value

Object of R6Class.

Methods

Profile$new(descriptor = descriptor) Use Profile.load to instantiate Profile class.
validate(descriptor) Validate a tabular data package descriptor against the Profile.
  • descriptor Retrieved and dereferenced tabular data package descriptor.
  • (Object) Returns TRUE if descriptor is valid or FALSE with error message.

Properties

name Returns profile name if available.
jsonschema Returns profile JSON Schema contents.

See Also

Profile Specifications

---

Profile.load Instantiate Profile class

Description

Constructor to instantiate Profile class.

Usage

Profile.load(profile)

Arguments

profile string profile name in registry or URL to JSON Schema

Value

Profile class object
### Readable

**Readable class**

- **Description**
  Readable class that allows typed access to its members

- **Usage**
  `Readable`

- **Format**
  `R6Class` object.

- **Value**
  Object of `R6Class`.

### ReadableArray

**ReadableArray class**

- **Description**
  Readable Array class

- **Usage**
  `ReadableArray`

- **Format**
  `R6Class` object.

- **Value**
  Object of `R6Class`.
ReadableConnection

Description
Readable connection class

Usage
ReadableConnection

Format

R6Class object.

Value
Object of R6Class.

Schema

Schema class

Description
A model of a schema with helpful methods for working with the schema and supported data. Schema instances can be initialized with a schema source as a url to a JSON file or a JSON object. The schema is initially validated (see validate). By default validation errors will be stored in $errors but in a strict mode it will be instantly raised.

Usage

# Schema.load(descriptor, strict=FALSE)

Arguments
descriptor schema descriptor, a JSON string, URL or file
strict flag to alter validation behaviour:
• if FALSE error will not be raised and all error will be collected in schema$errors
• if TRUE any validation error will be raised immediately

Format

R6Class object.
**Value**

Object of `R6Class`.

**Methods**

Schema$new(descriptor = descriptor, strict = strict) Use `Schema.load` to instantiate Schema class.

`getField(name)` Get schema field by name.

- `name` String with schema field name.
- (Field/NULL) Returns Field instance or NULL if not found.

`addField(descriptor)` Add new field to schema. The schema descriptor will be validated with newly added field descriptor.

- `descriptor` List of field descriptor.
- `TableSchemaError` Raises any error occurred in the process.
- (Field/NULL) Returns added Field instance or NULL if not added.

`removeField(name)` Remove field resource by name. The schema descriptor will be validated after field descriptor removal.

- `name` String with schema field name.
- `TableSchemaError` Raises any error occurred in the process.
- (Field/NULL) Returns removed Field instances or NULL if not found.

`castRow(row)` Cast row based on field types and formats.

- `row` Data row as a list of values.
- (any) Returns cast data row.

`infer(rows, headers=1)` Cast row based on field types and formats.

- `rows` List of lists representing rows.
- `headers` data sample headers, one of:
  - row number containing headers (rows should contain headers rows)
  - list of headers (rows should NOT contain headers rows)
- (Object) Returns Table Schema descriptor.

`commit(strict)` Cast row based on field types and formats.

- `strict` Boolean, alter strict mode for further work.
- `TableSchemaError` Raises any error occurred in the process.
- (Boolean) Returns TRUE on success and FALSE if not modified.

`save(target)` Cast row based on field types and formats.

- `target` String, path where to save a descriptor.
- `TableSchemaError` Raises any error occurred in the process.
- (Boolean) Returns TRUE on success.
Properties

valid  Returns validation status. It always TRUE in strict mode.
errors Returns validation errors. It always empty in strict mode.
descriptor Returns list of schema descriptor.
primaryKey  Returns string list of schema primary key.
foreignKeys Returns list of schema foreign keys.
fields  Returns list of Field instances.
fieldName  Returns a list of field names.

Language

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this package documents are to be interpreted as described in RFC 2119.

See Also

Schema.load, Table Schema Specifications

---

Schema.load  Instantiate Schema class

Description

Factory method to instantiate Schema class. This method is async and it should be used with value keyword from future package.

Usage

Schema.load(descriptor, strict=FALSE, caseInsensitiveHeaders = FALSE)

Arguments

descriptor  schema descriptor, a JSON string, URL or file
strict  flag to alter validation behaviour:
  • if FALSE error will not be raised and all error will be collected in schema$errors
  • if TRUE any validation error will be raised immediately

caseInsensitiveHeaders
  default is set to FALSE

Value

Schema class object
See Also

Schema, Table Schema Specifications

Examples

```r
SCHEMA <- '{
  "fields": [
    {"name": "id", "type": "string", "constraints": {"required": true}},
    {"name": "height", "type": "number"},
    {"name": "age", "type": "integer"},
    {"name": "name", "type": "string", "constraints": {"required": true}},
    {"name": "occupation", "type": "string"}
  ]
}'

# instantiate Schema class
def = Schema.load(descriptor = SCHEMA)
schema = future::value(def)

# correct number of fields
length(schema$fields)

# correct field names
schema$fieldNames

# convert row
row = list('string', '10.0', '1', 'string', 'string')
castRow = schema$castRow(row)
castRow

SCHEMA_MIN <- '{
  "fields": [
    {"name": "id"},
    {"name": "height"}
  ]
}'

# load schema
def2 = Schema.load(descriptor = SCHEMA_MIN)
schema2 = future::value(def2)

# set default types if not provided
schema2$fields[[1]]$type
schema2$fields[[2]]$type

# fields are not required by default
schema2$fields[[1]]$required
schema2$fields[[2]]$required

# work in strict mode
descriptor = '{"fields": [{"name": "name", "type": "string"}]}'
def3 = Schema.load(descriptor = descriptor, strict = TRUE)
schema3 = future::value(def3)
schema3$valid
```
Table class for working with data and schema.

Usage

# Table.load(source, schema = NULL, strict = FALSE, headers = 1, ...)

Table

<table>
<thead>
<tr>
<th>Table Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>

Table class for working with data and schema.
Arguments

- **source**: data source, one of:
  - string with the path of the local CSV file
  - string with the url of the remote CSV file
  - list of lists representing the rows
  - readable stream with CSV file contents
  - function returning readable stream with CSV file contents

- **schema**: data schema in all forms supported by Schema class

- **strict**: strictness option TRUE or FALSE, to pass to Schema constructor

- **headers**: data source headers, one of:
  - row number containing headers (source should contain headers rows)
  - list of headers (source should NOT contain headers rows)

... options to be used by CSV parser. All options listed at http://csv.adaltas.com/parse/#parser-options. By default `ltrim` is TRUE according to the CSV Dialect spec.

Format

- `R6Class` object.

Value

Object of `R6Class`.

Methods

- `Table$new(source, schema, strict, headers)`: Use `Table.load` to instantiate Table class.
- `iter(keyed, extended, cast=TRUE, relations=FALSE, stream=FALSE)`: Iter through the table data and emits rows cast based on table schema. Data casting could be disabled.
  - `keyed`: Iter keyed rows - TRUE/FALSE
  - `extended`: Iter extended rows - TRUE/FALSE
  - `cast`: Disable data casting if FALSE
  - `relations`: List object of foreign key references from a form of JSON `{resource1: [{field1: value1, field2: value2}, ...]}`. If provided foreign key fields will checked and resolved to its references
  - `stream`: Return Readable Stream of table rows if TRUE

- `read(keyed, extended, cast=TRUE, relations=FALSE, limit)`: Read the whole table and returns array of rows. Count of rows could be limited.
  - `keyed`: Flag to emit keyed rows - TRUE/FALSE
  - `extended`: Flag to emit extended rows - TRUE/FALSE
  - `cast`: Disable data casting if FALSE
  - `relations`: List object of foreign key references from a form of JSON `{resource1: [{field1: value1, field2: value2}, ...]}`. If provided foreign key fields will checked and resolved to its references
  - `limit`: Integer limit of rows to return if specified

- `infer(limit=100)`: Infer a schema for the table. It will infer and set Table Schema to `table$schema` based on table data.
Table

- limit - Limit rows sample size - number
  save(target) - Save data source to file locally in CSV format with , (comma) delimiter.
  - target - String path where to save a table data

Properties

headers - Returns data source headers
schema - Returns schema class instance

Details

A table is a core concept in a tabular data world. It represents a data with a metadata (Table Schema).
Tabular data consists of a set of rows. Each row has a set of fields (columns). We usually expect that each row has the same set of fields and thus we can talk about the fields for the table as a whole. In case of tables in spreadsheets or CSV files we often interpret the first row as a header row, giving the names of the fields. By contrast, in other situations, e.g. tables in SQL databases, the field names are explicitly designated.

In order to talk about the representation and processing of tabular data from text-based sources, it is useful to introduce the concepts of the physical and the logical representation of data.

The physical representation of data refers to the representation of data as text on disk, for example, in a CSV or JSON file. This representation may have some type information (JSON, where the primitive types that JSON supports can be used) or not (CSV, where all data is represented in string form).

The logical representation of data refers to the "ideal" representation of the data in terms of primitive types, data structures, and relations, all as defined by the specification. We could say that the specification is about the logical representation of data, as well as about ways in which to handle conversion of a physical representation to a logical one.

We'll explicitly refer to either the physical or logical representation in places where it prevents ambiguity for those engaging with the specification, especially implementors.

For example, constraints should be tested on the logical representation of data, whereas a property like missingValues applies to the physical representation of the data.

jsonlite package is internally used to convert json data to list objects. The input parameters of functions could be json strings, files or lists and the outputs are in list format to easily further process your data in R environment and exported as desired. More details about handling json you can see jsonlite documentation or vignettes here.

Future package is also used to load and create Table and Schema class asynchronously. To retrieve the actual result of the loaded Table or Schema you have to call value(future) to the variable you stored the loaded Table/Schema. More details about future package and sequential and parallel processing you can find here.

Examples section of each function show how to use jsonlite and future packages with tableschema.r.

Language

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this package documents are to be interpreted as described in RFC 2119.
See Also

Table.load, Table Schema Specifications

---

Table.load

*Instantiate Table class*

### Description

Factory method to instantiate Table class. This method is async and it should be used with `value` keyword from future package. If `references` argument is provided foreign keys will be checked on any reading operation.

### Usage

```r
Table.load(source, schema = NULL, strict = FALSE, headers = 1, ...)
```

### Arguments

- **source**: data source, one of:
  - string with the path of the local CSV file
  - string with the url of the remote CSV file
  - list of lists representing the rows
  - readable stream with CSV file contents
  - function returning readable stream with CSV file contents
- **schema**: data schema in all forms supported by Schema class
- **strict**: strictness option TRUE or FALSE, to pass to Schema constructor
- **headers**: data source headers, one of:
  - row number containing headers (source should contain headers rows)
  - list of headers (source should NOT contain headers rows)
- **...**: options to be used by CSV parser. All options listed at [http://csv.adaltas.com/parse/#parser-options](http://csv.adaltas.com/parse/#parser-options). By default `ltrim` is TRUE according to the CSV Dialect spec.

### Details

**jsonlite package** is internally used to convert json data to list objects. The input parameters of functions could be json strings, files or lists and the outputs are in list format to easily further process your data in R environment and exported as desired. Examples section show how to use jsonlite package and tableschema.r together. More details about handling json you can see jsonlite documentation or vignettes here.

**Future package** is also used to load and create Table and Schema classes asynchronously. To retrieve the actual result of the loaded Table or Schema you have to use `value` function to the variable you stored the loaded Table/Schema. More details about future package and sequential and parallel processing you can find here.

Term array refers to json arrays which if converted in R will be list objects.
See Also

Table, Table Schema Specifications

Examples

```python
# define source
SOURCE = '[
    "id", "height", "age", "name", "occupation"],
[1, 10.0, 1, "string!", "2012-06-15 00:00:00"],
[2, 10.1, 2, "string2", "2013-06-15 01:00:00"],
[3, 10.2, 3, "string3", "2014-06-15 02:00:00"],
[4, 10.3, 4, "string4", "2015-06-15 03:00:00"],
[5, 10.4, 5, "string5", "2016-06-15 04:00:00"]
]

# define schema
SCHEMA = '{
    "fields": [
        {"name": "id", "type": "integer", "constraints": {"required": true}},
        {"name": "height", "type": "number"},
        {"name": "age", "type": "integer"},
        {"name": "name", "type": "string", "constraints": {"unique": true}},
        {"name": "occupation", "type": "datetime", "format": "any"}
    ],
    "primaryKey": "id"
}'

def = Table.load(jsonlite::fromJSON(SOURCE, simplifyVector = FALSE), schema = SCHEMA)
table = future::value(def)

# work with list source
rows = table$read()

# read source data and limit rows
rows2 = table$read(limit = 1)

# read source data and return keyed rows
rows3 = table$read(limit = 1, keyed = TRUE)

# read source data and return extended rows
rows4 = table$read(limit = 1, extended = TRUE)

# work with Schema instance
def1 = Schema.load(SCHEMA)
schema = future::value(def1)
def2 = Table.load(jsonlite::fromJSON(SOURCE, simplifyVector = FALSE), schema = schema)
table2 = future::value(def2)
rows5 = table2$read()
TableSchemaError  

TableSchemaError class

Description

Error class for Table Schema

Usage

TableSchemaError

Format

R6Class object.

Value

Object of R6Class.

TRUE_VALUES  

default true values

Description

default true values

Usage

TRUE_VALUES

Format

An object of class character of length 4.
Types

Types class

Description

R6 class with Types and Formats.

*type* and *format* properties are used to give the type of the field (string, number etc) - see [types and formats](#) for more details. If type is not provided a consumer should assume a type of "string".

A field’s *type* property is a string indicating the type of this field.

A field’s *format* property is a string, indicating a format for the field type.

Both *type* and *format* are optional: in a field descriptor, the absence of a *type* property indicates that the field is of the type "string", and the absence of a *format* property indicates that the field’s *type* format is "default".

Types are based on the [type set of json-schema](#) with some additions and minor modifications (cf other type lists include those in Elasticsearch types).

Usage

Types

Format

*R6Class* object.

Value

Object of *R6Class*.

See Also

[Types and formats specifications](#), [types.castAny](#), [types.castBoolean](#), [types.castDate](#), [types.castDatetime](#), [types.castDuration](#), [types.castGejson](#), [types.castGeopoint](#), [types.castInteger](#), [types.castList](#), [types.castNumber](#), [types.castObject](#), [types.castString](#), [types.castTime](#), [types.castYear](#), [types.castYearmonth](#), [types.castArray](#)
types.castAny  

Cast any value

Description
Cast any value

Usage

types.castAny(format, value)

Arguments
format  

any format is accepted
value  

any value to cast

Details
Any type or format is accepted.

See Also
Types and formats specifications

Examples

types.castAny(format = "default", value = 1)
types.castAny(format = "default", value = "1")
types.castAny(format = "default", value = ")
types.castAny(format = "default", value = TRUE)

types.castArray  

Cast array

Description
Cast array is used for list objects

Usage

types.castArray(format, value)
types.castBoolean

Arguments

- format: no options (other than the default)
- value: lists, or valid JSON format arrays to cast

See Also

types.castList. Types and formats specifications

types.castBoolean

Description

Cast boolean values

Usage

types.castBoolean(format = "default", value, options = { })

Arguments

- format: no options (other than the default)
- value: boolean to cast
- options: specify additional true values or/and false values

Details

In the physical representations of data where boolean values are represented with strings, the values set in trueValues and falseValues are to be cast to their logical representation as booleans. trueValues and falseValues are lists which can be customised to user need. The default values for these are in the additional properties section below.

The boolean field can be customised with these additional properties:

- trueValues: ["true", "True", "TRUE", "1"]
- falseValues: ["false", "False", "FALSE", "0"]

See Also

Types and formats specifications
Examples

types.castBoolean(format = "default", value = TRUE)

types.castBoolean(format = "default", value = "true")

types.castBoolean(format = "default", value = "1")

types.castBoolean(format = "default", value = "0")

# set options with additional true value
types.castBoolean(format = "default", value = "yes", list(trueValues = list("yes")))

# set options with additional false value
types.castBoolean(format = "default", value = "no", list(falseValues = list("no")))

---

**types.castDate**

*Cast date*

description

cast date without a time

Usage

types.castDate(format = "default", value)

Arguments

- **format**
  - available options are "default", "any", and "<pattern>" where
  - default: An ISO8601 format string
    - date: This **MUST** be in ISO8601 format YYYY-MM-DD
    - datetime: a date-time. This **MUST** be in ISO 8601 format of YYYY-MM-DDThh:mm:ssZ in UTC time
    - time: a time without a date
  - any: Any parsable representation of the type. The implementing library can attempt to parse the datetime via a range of strategies, e.g. lubridate, parseDate, strptime, DateTimeClasses.
  - <pattern>: date/time values in this field can be parsed according to pattern. <pattern> **MUST** follow the syntax of strptime. (That is, values in this field should be parseable by R using <pattern>.)

- **value**: date to cast

See Also

Types and formats specifications, strptime, DateTimeClasses, parsedate-package and lubridate-package.
**Examples**

```r
types.castDate(format = "default", value = as.Date("2019-1-1"))
```

```r
types.castDate(format = "default", value = "2019-1-1")
```

```r
types.castDate(format = "any", value = "2019-1-1")
```

```r
types.castDate(format = "%d/%m/%y", value = "21/11/06")
```

```r
types.castDate(format = "%d/%m/%y", value = as.Date("2019-1-1"))
```

---

**types.castDatetime**

**Cast datetime**

**Description**

Cast date with time

**Usage**

```r
types.castDatetime(format = "%Y-%m-%dT%H:%M:%SZ", value)
```

**Arguments**

- **format**
  
  available options are "default", "any", and "<pattern>" where

  - **default**
    
    An ISO8601 format string e.g. YYYY-MM-DDThh:mm:ssZ in UTC

  - **any**
    
    As for types.castDate

  - **<pattern>**
    
    As for types.castDate

- **value**
  
  datetime to cast

**See Also**

Types and formats specifications, `strptime`, `DateTimeClasses`, `parsedate-package` and `lubridate-package`.

**Examples**

```r
types.castDatetime(format = "default", value = "2014-01-01T06:00:00Z")
```

```r
types.castDatetime(format = "%d/%m/%y %H:%M", value = "21/11/06 16:30")
```
**types.castDuration**

*Cast duration of time*

---

**Description**

Cast duration of time

**Usage**

```r
types.castDuration(format = "default", value)
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>no options (other than the default)</td>
</tr>
<tr>
<td>value</td>
<td>duration to cast</td>
</tr>
</tbody>
</table>

**Details**

We follow the definition of XML Schema duration datatype directly and that definition is implicitly inlined here.

To summarize: the lexical representation for duration is the ISO 8601 extended format `PnYnMnDTnHnMnSnS`, where `nY` represents the number of years, `nM` the number of months, `nD` the number of days, `'T'` is the date/time separator, `nH` the number of hours, `nM` the number of minutes and `nS` the number of seconds. The number of seconds can include decimal digits to arbitrary precision. Date and time elements including their designator may be omitted if their value is zero, and lower order elements may also be omitted for reduced precision.

**See Also**

Types and formats specifications, lubridate-package.

**Examples**

```r
types.castDuration(format = "default", value = durations(years=10))
```

```r
types.castDuration(format = "default", value = "P1Y10M3DT5H11M7S")
```
**types.castGeojson**

Cast JSON object according to GeoJSON or TopoJSON spec

**Description**

Cast JSON object according to GeoJSON or TopoJSON spec

**Usage**

```javascript
types.castGeojson(format, value)
```

**Arguments**

- **format**
  - `default` is a geojson object as per the GeoJSON spec or topojson object as per the TopoJSON spec
- **value**
  - GeoJSON to cast

**See Also**

Types and formats specifications

---

**types.castGeopoint**

Cast geographic point

**Description**

Cast geographic point

**Usage**

```javascript
types.castGeopoint(format, value)
```

**Arguments**

- **format**
  - available options are "default", "array" and "object", where
    - `default` A string of the pattern "lon, lat", where lon is the longitude and lat is the latitude (note the space is optional after the ,). E.g. "90, 45".
    - `array` A JSON array, or a string parsable as a JSON array, of exactly two items, where each item is a number, and the first item is lon and the second item is lat e.g. [90, 45].
    - `object` A JSON object with exactly two keys, lat and lon and each value is a number e.g. {"lon": 90, "lat": 45}.
- **value**
  - geopoint to cast
See Also

Types and formats specifications

Examples

```r
types.castGeopoint(format = "default", value = list(180, 90))
types.castGeopoint(format = "default", value = '180,90')
types.castGeopoint(format = "default", value = '180, -90')
types.castGeopoint(format = "array", value = list(180, 90))
types.castGeopoint(format = "array", value = '[180, -90]')
types.castGeopoint(format = "object", value = list(lon = 180, lat = 90))
types.castGeopoint(format = "object", value = '{"lon": 180, "lat": 90}')
```

types.castInteger Cast integer

Description

Cast integer. Integer values are indicated in the standard way for any valid integer.

Usage

```r
types.castInteger(format, value, options = list())
```

Arguments

- **format**: no options (other than the default)
- **value**: integer to cast
- **options**: named list set bareNumber TRUE or FALSE, see details

Details

`bareNumber` is a boolean field with a default of TRUE. If TRUE the physical contents of this field must follow the formatting constraints already set out. If FALSE the contents of this field may contain leading and or trailing non-numeric characters (which implementors MUST therefore strip). The purpose of `bareNumber` is to allow publishers to publish numeric data that contains trailing characters such as percentages e.g. 95 if anything, they do with stripped text.
See Also

Types and formats specifications

Examples

types.castInteger(format = "default", value = 1)

types.castInteger(format = "default", value = "1")
# cast trailing non numeric character
types.castInteger(format = "default", value = "1$", options = list(bareNumber = FALSE))

types.castList(format, value)

Arguments

format no options (other than the default)
value lists, or valid JSON format arrays to cast

See Also

Types and formats specifications

Examples

types.castList(format = "default", value = list())

types.castList(format = "default", value = list('key', 'value'))

types.castList(format = "default", value = ['"key", "value"']) # cast valid json array
types.castNumber  Cast numbers of any kind including decimals

Description
Cast numbers of any kind including decimals.

Usage
	types.castNumber(format, value, options = {})

Arguments
format no options (other than the default)
value number to cast
options available options are "decimalChar", "groupChar" and "bareNumber", where
decimalChar A string whose value is used to represent a decimal point within
the number. The default value is ".".
groupChar A string whose value is used to group digits within the number.
The default value is "," e.g. "100,000".
bareNumber A boolean field with a default of TRUE. If TRUE the physical contents
of this field must follow the formatting constraints already set out. If FALSE the contents
of this field may contain leading and/or trailing non-numeric characters (which implementors
MUST therefore strip). The purpose of bareNumber is to allow publishers to publish numeric
data that contains trailing characters such as percentages e.g. 95 e.g. €95 or EUR 95. Note
that it is entirely up to implementors what, if anything, they do with stripped text.

Details
The lexical formatting follows that of decimal in XMLSchema: a non-empty finite-length sequence
of decimal digits separated by a period as a decimal indicator. An optional leading sign is allowed.
If the sign is omitted, "+" is assumed. Leading and trailing zeroes are optional. If the fractional part
is zero, the period and following zero(es) can be omitted. For example: '-1.23', '12678967.543233',
'+100000.00', '210'.
The following special string values are permitted (case need not be respected):

• NaN: not a number
• INF: positive infinity
• -INF: negative infinity

A number MAY also have a trailing:

• exponent: this MUST consist of an E followed by an optional + or - sign followed by one or
  more decimal digits (0-9)
**types.castObject**

**Description**

Cast object data which is lists or valid JSON.

**Usage**

`types.castObject(format, value)`

**Arguments**

- `format` : no options (other than the default)
- `value` : object to cast

**See Also**

Types and formats specifications

**Examples**

```bash
types.castObject(format = "default", value = list())

types.castObject(format = "default", value = "{}")

types.castObject(format = "default", value = "{'key': "value"}")
```
**types.castString**  
*Cast string*

**Description**

Cast string that is, sequences of characters.

**Usage**

`types.castString(format, value)`

**Arguments**

- **format**
  
  available options are "default", "email", "uri", "binary" and "uuid", where
  
  - default  Any valid string.
  - email  A valid email address.
  - uri  A valid URI.
  - binary  A base64 encoded string representing binary data.
  - uuid  A string that is a uuid.

- **value**
  
  string to cast

**See Also**

Types and formats specifications

**Examples**

```
# cast any string
types.castString(format = "default", value = "string")

# cast email
types.castString(format = "email", value = "name@gmail.com")

# cast binary
types.castString(format = "binary", value = "dGVzdA==")

# cast uuid
types.castString(format = "uuid", value = "95ecc380-afe9-11e4-9b6c-751b66dd541e")
```
**types.castTime**

*Cast time without a date*

**Description**

Cast time without a date.

**Usage**

```
types.castTime(format = "%H:%M:%S", value)
```

**Arguments**

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>format</code></td>
<td>available options are &quot;default&quot;, &quot;any&quot;, and &quot;&lt;pattern&gt;&quot; where</td>
</tr>
<tr>
<td><code>default</code></td>
<td>An ISO8601 time string e.g. hh:mm:ss</td>
</tr>
<tr>
<td><code>any</code></td>
<td>As for <code>types.castDate</code></td>
</tr>
<tr>
<td><code>&lt;pattern&gt;</code></td>
<td>As for <code>types.castDate</code></td>
</tr>
<tr>
<td><code>value</code></td>
<td>time to cast</td>
</tr>
</tbody>
</table>

**See Also**

Types and formats specifications, `strptime`, `DateTimeClasses`, `parsedate-package` and `lubridate-package`.

**Examples**

```
types.castTime(format = "default", value = '06:00:00')
```

---

**types.castYear**

*Cast year*

**Description**

Cast year. A calendar year as per XMLSchema gYear. Usual lexical representation is: YYYY.

**Usage**

```
types.castYear(format, value)
```

**Arguments**

<table>
<thead>
<tr>
<th>Format</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>format</code></td>
<td>no options (other than the default)</td>
</tr>
<tr>
<td><code>value</code></td>
<td>year to cast</td>
</tr>
</tbody>
</table>
types.castYearmonth

**See Also**

Types and formats specifications

**Examples**

```python
types.castYear(format = "default", value = 2000)
types.castYear(format = "default", value = "2010")
```

---

**types.castYearmonth**  
*Cast a specific month in a specific year*

**Description**

Cast a specific month in a specific year as per XMLSchema gYearMonth. Usual lexical representation is: YYYY-MM.

**Usage**

```python
types.castYearmonth(format, value)
```

**Arguments**

- `format`: no options (other than the default)
- `value`: list or string with yearmonth to cast

**See Also**

Types and formats specifications

**Examples**

```python
types.castYearmonth(format = "default", value = list(2000, 10))
types.castYearmonth(format = "default", value = "2018-11")
```
validate validate descriptor

Description
Validates whether a schema is a validate Table Schema accordingly to the specifications. It does not validate data against a schema.

Usage
validate(descriptor)

Arguments
descriptor schema descriptor, one of:
- string with the local CSV file (path)
- string with the remote CSV file (url)
- list object

Value
TRUE on valid

Writeable Writeable class

Description
Writable streams class

Usage
Writeable

Format
R6Class object.

Value
Object of R6Class.
write_json  

**Save json file**

**Description**

save json

**Usage**

`write_json(x, file)`

**Arguments**

```
x       list object
file    file
```
Index

* Topic **datasets**
  - DEFAULT_DECIMAL_CHAR, 13
  - DEFAULT_GROUP_CHAR, 14
  - FALSE_VALUES, 15
  - TRUE_VALUES, 36

* Topic **data**
  - Constraints, 7
  - Field, 15
  - Profile, 24
  - Readable, 26
  - ReadableArray, 26
  - ReadableConnection, 27
  - Schema, 27
  - Table, 31
  - TableSchemaError, 36
  - Types, 37
  - Writeable, 51

Constraints, 4, 7
- constraints.checkEnum, 7, 7
- constraints.checkMaximum, 7, 8
- constraints.checkMaxLength, 7, 9
- constraints.checkMinimum, 7, 10
- constraints.checkMinLength, 7, 10
- constraints.checkPattern, 7, 11
- constraints.checkRequired, 7, 12
- constraints.checkUnique, 7, 13

DateTimeClasses, 40, 41, 49
- DEFAULT_DECIMAL_CHAR, 13
- DEFAULT_GROUP_CHAR, 14
- durations, 14

FALSE_VALUES, 15
- Field, 4, 15

helpers.expandFieldDescriptor, 18
- helpers.expandSchemaDescriptor, 18
- helpers.from.json.to.list, 19
- helpers.from.list.to.json, 19
- helpers.retrieveDescriptor, 19
- infer, 20
- is.binary, 21
- is.email, 21
- is.uri, 22
- is.uuid, 22
- is.valid, 23
- is_empty, 23
- is_integer, 24
- is_object, 24

Profile, 24, 25
- Profile.load, 25, 25
- R6Class, 7, 15, 16, 25–28, 32, 36, 37, 51
- Readable, 26
- ReadableArray, 26
- ReadableConnection, 27

Schema, 27
- Schema.load, 28, 29, 29
- strptime, 40, 41, 49

Table, 31, 35
- Table.load, 20, 20, 32, 34, 34
- tableschema.r-package, 3
- TableSchemaError, 36
- TRUE_VALUES, 36
- Types, 4, 37
- types.castAny, 37, 38
- types.castArray, 37, 38
- types.castBoolean, 37, 39
- types.castDate, 37, 40, 41, 49
- types.castDatetime, 37, 41
- types.castDuration, 14, 37, 42
- types.castGeojson, 37, 43
- types.castGeopoint, 37, 43
- types.castInteger, 37, 44
- types.castList, 37, 39, 45
- types.castNumber, 37, 46
types.castObject, 37, 47
types.castString, 37, 48
types.castTime, 37, 49
types.castYear, 37, 49
types.castYearmonth, 37, 50

validate, 51
value, 6, 29, 34

write_json, 52
Writeable, 51