Package ‘datapackage.r’

October 11, 2021

Type Package
Title Data Package 'Frictionless Data'
Version 1.3.4
Date 2021-10-11
Maintainer Kleanthis Koupidis <koupidis.okfgr@gmail.com>
Description Work with 'Frictionless Data Packages' (<https://specs.frictionlessdata.io//data-package/>). Allows to load and validate any descriptor for a data package profile, create and modify descriptors and provides expose methods for reading and streaming data in the package. When a descriptor is a 'Tabular Data Package', it uses the 'Table Schema' package (<https://CRAN.R-project.org/package=tableschema.r>) and exposes its functionality, for each resource object in the resources field.
URL https://github.com/frictionlessdata/datapackage-r
License MIT + file LICENSE
BugReports https://github.com/frictionlessdata/datapackage-r/issues
Encoding UTF-8
Imports config, future, httr, iterators, jsonlite, jsonvalidate, purrr, R6, R.utils, readr, rlist, stringr, tableschema.r, tools, urltools, utils, V8
Suggests covr, curl, data.table, DBI, devtools, foreach, httptest, knitr, rmarkdown, RSQLite, testthat
Collate 'DataPackageError.R' 'Package.R' 'helpers.R' 'profile.R'
  'binary.readable.connection.r' 'binary.readable.r'
  'datapackage.r.R' 'infer.R' 'is.valid.R' 'resource.R'
  'validate.R'
RoxygenNote 7.1.2
VignetteBuilder knitr
NeedsCompilation no
Author Kleanthis Koupidis [aut, cre],
  Lazaros Ioannidis [aut],
  Charalampos Bratsas [aut],
  Open Knowledge International [cph]
Repository: CRAN
Date/Publication: 2021-10-11 18:50:08 UTC

**R topics documented:**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>datapackage.r-package</td>
<td>3</td>
</tr>
<tr>
<td>BinaryReadable</td>
<td>7</td>
</tr>
<tr>
<td>BinaryReadableConnection</td>
<td>8</td>
</tr>
<tr>
<td>DataPackageError</td>
<td>9</td>
</tr>
<tr>
<td>dereferencePackageDescriptor</td>
<td>10</td>
</tr>
<tr>
<td>dereferenceResourceDescriptor</td>
<td>10</td>
</tr>
<tr>
<td>descriptor.pointer</td>
<td>11</td>
</tr>
<tr>
<td>expandPackageDescriptor</td>
<td>11</td>
</tr>
<tr>
<td>expandResourceDescriptor</td>
<td>11</td>
</tr>
<tr>
<td>file_basename</td>
<td>12</td>
</tr>
<tr>
<td>file_extension</td>
<td>12</td>
</tr>
<tr>
<td>filterDefaultDialect</td>
<td>12</td>
</tr>
<tr>
<td>findFiles</td>
<td>13</td>
</tr>
<tr>
<td>helpers.from.json.to.list</td>
<td>13</td>
</tr>
<tr>
<td>helpers.from.list.to.json</td>
<td>14</td>
</tr>
<tr>
<td>infer</td>
<td>14</td>
</tr>
<tr>
<td>is.compressed</td>
<td>15</td>
</tr>
<tr>
<td>is.empty</td>
<td>15</td>
</tr>
<tr>
<td>is.git</td>
<td>16</td>
</tr>
<tr>
<td>is.json</td>
<td>16</td>
</tr>
<tr>
<td>is.local.descriptor.path</td>
<td>17</td>
</tr>
<tr>
<td>is.valid</td>
<td>17</td>
</tr>
<tr>
<td>isRemotePath</td>
<td>18</td>
</tr>
<tr>
<td>isSafePath</td>
<td>18</td>
</tr>
<tr>
<td>isUndefined</td>
<td>19</td>
</tr>
<tr>
<td>locateDescriptor</td>
<td>19</td>
</tr>
<tr>
<td>Package</td>
<td>20</td>
</tr>
<tr>
<td>Package.load</td>
<td>23</td>
</tr>
<tr>
<td>Profile</td>
<td>25</td>
</tr>
<tr>
<td>Profile.load</td>
<td>26</td>
</tr>
<tr>
<td>push</td>
<td>27</td>
</tr>
<tr>
<td>Resource</td>
<td>27</td>
</tr>
<tr>
<td>Resource.load</td>
<td>30</td>
</tr>
<tr>
<td>retrieveDescriptor</td>
<td>32</td>
</tr>
<tr>
<td>validate</td>
<td>33</td>
</tr>
<tr>
<td>validateDialect</td>
<td>34</td>
</tr>
<tr>
<td>write.json</td>
<td>34</td>
</tr>
</tbody>
</table>

Index: 35
Description

Work frictionless with 'Data Packages' (<https://specs.frictionlessdata.io/data-package/>). Allows to load and validate any descriptor for a data package profile, create and modify descriptors and provides expose methods for reading and streaming data in the package. When a descriptor is a 'Tabular Data Package', it uses the 'Table Schema' package (<https://CRAN.R-project.org/package=tableschema.r>) and exposes its functionality, for each resource object in the resources field.

Introduction

A Data Package consists of:

- Metadata that describes the structure and contents of the package.
- Resources such as data files that form the contents of the package.

The Data Package metadata is stored in a "descriptor". This descriptor is what makes a collection of data a Data Package. The structure of this descriptor is the main content of the specification below.

In addition to this descriptor a data package will include other resources such as data files. The Data Package specification does NOT impose any requirements on their form or structure and can therefore be used for packaging any kind of data.

The data included in the package may be provided as:

- Files bundled locally with the package descriptor.
- Remote resources, referenced by URL.
- "Inline" data which is included directly in the descriptor.

**Jsolite 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. It is recommended to use `helpers.from.json.to.list` or `helpers.from.list.to.json` to convert json objects to lists and vice versa. More details about handling json you can see jsonlite documentation or vignettes here.

Several example data packages can be found in the datasets organization on github, including:

- World GDP
- ISO 3166-2 country codes

Specification

#
Descriptor

The descriptor is the central file in a Data Package. It provides:

- General metadata such as the package’s title, license, publisher etc
- A list of the data "resources" that make up the package including their location on disk or online and other relevant information (including, possibly, schema information about these data resources in a structured form)

A Data Package descriptor MUST be a valid JSON object. (JSON is defined in RFC 4627). When available as a file it MUST be named datapackage.json and it MUST be placed in the top-level directory (relative to any other resources provided as part of the data package).

The descriptor MUST contain a resources property describing the data resources.

All other properties are considered metadata properties. The descriptor MAY contain any number of other metadata properties. The following sections provides a description of required and optional metadata properties for a Data Package descriptor.

Adherence to the specification does not imply that additional, non-specified properties cannot be used: a descriptor MAY include any number of properties in addition to those described as required and optional properties. For example, if you were storing time series data and wanted to list the temporal coverage of the data in the Data Package you could add a property temporal (cf Dublin Core):

```
"temporal": { "name": "19th Century","start": "1800-01-01","end": "1899-12-31" }
```

This flexibility enables specific communities to extend Data Packages as appropriate for the data they manage. As an example, the Tabular Data Package specification extends Data Package to the case where all the data is tabular and stored in CSV.

Resource Information

Packaged data resources are described in the resources property of the package descriptor. This property MUST be an array/list of objects. Each object MUST follow the Data Resource specification.

See also Resource Class

Metadata

#

Required Properties

The resources property is required, with at least one resource.

Recommended Properties

In addition to the required properties, the following properties SHOULD be included in every package descriptor:

name A short url-usable (and preferably human-readable) name of the package. This MUST be lower-case and contain only alphanumeric characters along with ",", ",_" or ",-" characters. It will
function as a unique identifier and therefore SHOULD be unique in relation to any registry in which this package will be deposited (and preferably globally unique).

The name SHOULD be invariant, meaning that it SHOULD NOT change when a data package is updated, unless the new package version should be considered a distinct package, e.g. due to significant changes in structure or interpretation. Version distinction SHOULD be left to the version property. As a corollary, the name also SHOULD NOT include an indication of time range covered.

id A property reserved for globally unique identifiers. Examples of identifiers that are unique include UUIDs and DOIs.

A common usage pattern for Data Packages is as a packaging format within the bounds of a system or platform. In these cases, a unique identifier for a package is desired for common data handling workflows, such as updating an existing package. While at the level of the specification, global uniqueness cannot be validated, consumers using the id property MUST ensure identifiers are globally unique.

Examples:
- {"id": "b03ec84-77fd-4270-813b-0c698943f7ce"}
- {"id": "https://doi.org/10.1594/PANGAEA.726855"}

licenses The license(s) under which the package is provided.

This property is not legally binding and does not guarantee the package is licensed under the terms defined in this property.

licenses MUST be an array. Each item in the array is a License. Each MUST be an object. The object MUST contain a name property and/or a path property. It MAY contain a title property.

Here is an example:
"licenses": [ { "name": "ODC-PDDL-1.0","path": "https://opendatacommons.org/licenses/pddl/","title": "Open Data Commons Public Domain Dedication and License v1.0" } ]

- name: The name MUST be an Open Definition license ID.
- path: A url-or-path string, that is a fully qualified HTTP address, or a relative POSIX path (see the url-or-path definition in Data Resource for details).

profile A string identifying the profile of this descriptor as per the profiles specification.

Examples:
- {"profile": "tabular-data-package"}
- {"profile": "https://example.com/my-profiles-json-schema.json"}

Optional Properties

The following are commonly used properties that the package descriptor MAY contain:

title A string providing a title or one sentence description for this package.
description A description of the package. The description MUST be markdown formatted – this also allows for simple plain text as plain text is itself valid markdown. The first paragraph (up to the first double line break) should be usable as summary information for the package.
hompage A URL for the home on the web that is related to this data package.
version A version string identifying the version of the package. It should conform to the Semantic Versioning requirements and should follow the Data Package Version pattern.
sources The raw sources for this data package. It MUST be an array of Source objects. Each Source object MUST have a title and MAY have path and/or email properties.
Example:
"sources": [{"title": "World Bank and OECD","path": "https://data.worldbank.org/indicator/NY.GDP.MKTP.CD"}]
  • title: Title of the source (e.g. document or organization name).
  • path: A url-or-path string, that is a fully qualified HTTP address, or a relative POSIX path (see the url-or-path definition in Data Resource for details).
  • email: An email address.

contributors The people or organizations who contributed to this Data Package. It MUST be an array. Each entry is a Contributor and MUST be an object. A Contributor MUST have a title property and MAY contain path, email, role and organization properties. An example of the object structure is as follows:
Example:
"contributors": [{"title": "Joe Bloggs","email": "joe@bloggs.com","path": "https://www.bloggs.com","author"}]
  • title: Name/Title of the contributor (name for person, name/title of organization).
  • path: A fully qualified https URL pointing to a relevant location online for the contributor.
  • email: An email address.
  • role: A string describing the role of the contributor. It MUST be one of: author, publisher, maintainer, wrangler, and contributor. Defaults to contributor.
  – Note on semantics: use of the "author" property does not imply that that person was the original creator of the data in the data package - merely that they created and/or maintain the data package. It is common for data packages to "package" up data from elsewhere. The original origin of the data can be indicated with the sources property - see above.
  • organization: A string describing the organization this contributor is affiliated to.

details

Jsolite 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. It is recommended to use helpers.from.json.to.list
or `helpers.from.list.to.json` to convert json objects to lists and vice versa. More details about handling json you can see jsonlite documentation or vignettes here. 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**

Data Package Specifications

---

<table>
<thead>
<tr>
<th>BinaryReadable</th>
<th>Readable class</th>
</tr>
</thead>
</table>

**Description**

Readable class

**Format**

`R6Class` object.

**Value**

Object of `R6Class`.

**Methods**

**Public methods:**

- `BinaryReadable$new()`
- `BinaryReadable$read()`
- `BinaryReadable$clone()`

**Method `new()`:**

*Usage:*

`BinaryReadable$new(options = list())`

**Method `read()`:**

*Usage:*

`BinaryReadable$read(size = NULL)`

**Method `clone()`:** The objects of this class are cloneable with this method.

*Usage:*

`BinaryReadable$clone(deep = FALSE)`

*Arguments:*

- `deep` Whether to make a deep clone.
BinaryReadableConnection

**Binary Readable Connection class**

---

**Description**

Binary Readable connection class

**Format**

R6Class object.

**Value**

Object of R6Class.

**Methods**

**Public methods:**

- `BinaryReadableConnection$new()`
- `BinaryReadableConnection$read()`
- `BinaryReadableConnection$clone()`

**Method `new()`:**

*Usage:*

`BinaryReadableConnection$new(options = list())`

**Method `read()`:**

*Usage:*

`BinaryReadableConnection$read(size = NULL)`

**Method `clone()`:** The objects of this class are cloneable with this method.

*Usage:*

`BinaryReadableConnection$clone(deep = FALSE)`

*Arguments:*

- `deep` Whether to make a deep clone.
**DataPackageError**

**DataPackageError class**

### Description

Error class for Data Package

### Format

*R6Class* object.

### Value

Object of *R6Class*.

### Super class

`tableschema.r::TableSchemaError` -> DataPackageError

### Methods

**Public methods:**

- `DataPackageError$new()`
- `DataPackageError$clone()`

**Method new():**

*Usage:*

DataPackageError$new(message, error = NULL)

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

DataPackageError$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.
dereferencePackageDescriptor

Dereference package descriptor

Description
Helper function to dereference package descriptor.

Usage
dereferencePackageDescriptor(descriptor, basePath)

Arguments
- descriptor: descriptor
- basePath: basePath

dereferenceResourceDescriptor

Dereference resource descriptor

Description
Helper function to dereference resource descriptor.

Usage
dereferenceResourceDescriptor(descriptor, basePath, baseDescriptor = NULL)

Arguments
- descriptor: descriptor
- basePath: basePath
- baseDescriptor: baseDescriptor
**descriptor.pointer**  
*Descriptor pointer*

**Description**  
Helper function for specifying locations in a descriptor.

**Usage**  
descriptor.pointer(value, baseDescriptor)

**Arguments**  
- value: value that specifies location in the descriptor  
- baseDescriptor: base descriptor

---

**expandPackageDescriptor**  
*Expand package descriptor*

**Description**  
Helper function to expand package descriptor.

**Usage**  
expandPackageDescriptor(descriptor)

**Arguments**  
- descriptor: descriptor

---

**expandResourceDescriptor**  
*Expand resource descriptor*

**Description**  
Helper function to expand resource descriptor.

**Usage**  
expandResourceDescriptor(descriptor)

**Arguments**  
- descriptor: descriptor
**file basename**  
*File basename*

**Description**
Removes all of the path up to and including the last path separator (if any) without extensions.

**Usage**
`file_basename(path)`

**Arguments**
- `path` character vector with path names

---

**file extension**  
*File extension*

**Description**
Returns the file extension without the leading dot.

**Usage**
`file_extension(path)`

**Arguments**
- `path` string with path names

---

**filterDefaultDialect**  
*Filter Default Dialect*

**Description**
Helper function to filter default dialect `quoteChar` and `escapeChar` are mutually exclusive: https://frictionlessdata.io/specs/csv-dialect/#specification

**Usage**
`filterDefaultDialect(dialect = NULL)`

**Arguments**
- `dialect` list
findFiles

Value
dialect list

Description
Find a file pattern in a specified directory.

Usage
findFiles(pattern, path = getwd())

Arguments
- pattern: string pattern
- path: string path

helpers.from.json.to.list

Convert json to list

Description
Helper function convert json to list.

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

Arguments
- lst: list object
helpers.from.list.to.json

*Convert list to json*

**Description**

Helper function convert list to json.

**Usage**

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

**Arguments**

- **json**: json string

---

**infer**

*Infer a data package descriptor*

**Description**

A standalone function to infer a data package descriptor.

**Usage**

infer(pattern = NULL, basePath = NULL)

**Arguments**

- **pattern**: string with file pattern
- **basePath**: base path for all relative paths

**Value**

Data Package Descriptor

**Examples**

```r
## Not run:
descriptor = infer("csv",basePath = ".")
descriptor

## End(Not run)
```
### is.compressed

**Description**
Helper function to check if a file is compressed.

**Usage**

```r
is.compressed(x)
```

**Arguments**

- `x` string with the file’s path

**Value**

TRUE if file is compressed

### is.empty

**Description**
Is empty list

**Usage**

```r
is.empty(list)
```

**Arguments**

- `list` list

**Value**

TRUE if list is empty
is.git  

Description
Helper function to check if a link is from git.

Usage
is.git(x)

Arguments
x url

Value
TRUE if url is git

is.json  

Description
Check if an object is json.

Usage
is.json(object)

Arguments
object object to test if it’s json

Value
TRUE if object is json
is.local.descriptor.path

Is Local Descriptor Path

Description

Helper function to check if a descriptor is local

Usage

is.local.descriptor.path(descriptor, directory = ".")

Arguments

descriptor descriptor

directory A character vector of full path name. The default corresponds to the working
directory specified by getwd

is.valid 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
isRemotePath  
*Is remote path*

**Description**  
Helper function to identify a remote path.

**Usage**  
isRemotePath(path)

**Arguments**  
```markdown
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>string path</td>
</tr>
</tbody>
</table>
```

**Value**  
TRUE if path is remote

---

isSafePath  
*Is safe path*

**Description**  
Helper function to check if a path is safe.

**Usage**  
isSafePath(path)

**Arguments**  
```markdown
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>string path</td>
</tr>
</tbody>
</table>
```

**Value**  
TRUE if path is safe
**isUndefined**

*Check if a variable is undefined or NULL*

---

**Description**

Helper function to check if a variable is undefined or NULL.

**Usage**

```javascript
isUndefined(x)
```

**Arguments**

- `x` variable

**Value**

TRUE if variable is undefined

---

**locateDescriptor**

*Locate descriptor*

---

**Description**

Helper function to locate descriptor.

**Usage**

```javascript
locateDescriptor(descriptor)
```

**Arguments**

- `descriptor` descriptor
**Package**  

Data Package class

**Description**

A class for working with data packages. It provides various capabilities like loading local or remote data package, inferring a data package descriptor, saving a data package descriptor and many more.

**Usage**

```r
# Package.load(descriptor = list(), basePath = NA, strict = FALSE)
```

**Format**

R6Class object

**Value**

Object of R6Class

**Methods**

- `Package$new(descriptor = list(), basePath = NA, strict = FALSE)` Use `Package.load` to instantiate Package class.
- `getResource(name)` Get data package resource by name or null if not found.
  - name Data resource name.
- `addResource(descriptor)` Add new resource to data package. The data package descriptor will be validated with newly added resource descriptor.
  - descriptor Data resource descriptor.
- `removeResource(name)` Remove data package resource by name. The data package descriptor will be validated after resource descriptor removal.
  - name Data resource name.
- `infer(pattern=FALSE)` Infer a data package metadata. If `pattern` is not provided only existent resources will be inferred (added metadata like encoding, profile etc). If `pattern` is provided new resources with file names matching the pattern will be added and inferred. It commits changes to data package instance.
  - pattern Glob pattern for new resources.
- `commit(strict)` Update data package instance if there are in-place changes in the descriptor. Returns TRUE on success and FALSE if not modified.
  - strict Boolean - Alter strict mode for further work.
- `save(target)` For now only descriptor will be saved. Save descriptor to target destination.
  - target String path where to save a data package.
Properties

- **valid** Returns validation status. It always TRUE in strict mode.
- **errors** Returns validation errors. It always empty in strict mode.
- **profile** Returns an instance of `Profile` class.
- **descriptor** Returns list of package descriptor.
- **resources** Returns list of Resource instances.
- **resourceNames** Returns list of resource names.

Details

A Data Package consists of:

- Metadata that describes the structure and contents of the package.
- Resources such as data files that form the contents of the package.

The Data Package metadata is stored in a "descriptor". This descriptor is what makes a collection of data a Data Package. The structure of this descriptor is the main content of the specification below. In addition to this descriptor a data package will include other resources such as data files. The Data Package specification does NOT impose any requirements on their form or structure and can therefore be used for packaging any kind of data.

The data included in the package may be provided as:

- Files bundled locally with the package descriptor.
- Remote resources, referenced by URL.
- "Inline" data which is included directly in the descriptor.

**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. It is recommended to use `helpers.from.json.to.list` or `helpers.from.list.to.json` to convert json objects to lists and vice versa. More details about handling json you can see jsonlite documentation or vignettes here.

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**.

Methods

**Public methods:**

- `Package$new()`
- `Package$addResource()`
- `Package$getResource()`
- `Package$removeResource()`
- `Package$infer()`
- `Package$commit()`
• Package$save()
• Package$clone()

Method new():
Usage:
Package$new(
  descriptor = list(),
  basePath = NULL,
  strict = FALSE,
  profile = NULL
)

Method addResource():
Usage:
Package$addResource(descriptor)

Method getResource():
Usage:
Package$getResource(name)

Method removeResource():
Usage:
Package$removeResource(name)

Method infer():
Usage:
Package$infer(pattern)

Method commit():
Usage:
Package$commit(strict = NULL)

Method save():
Usage:
Package$save(target, type = "json")

Method clone(): The objects of this class are cloneable with this method.
Usage:
Package$clone(deep = FALSE)
Arguments:
depth Whether to make a deep clone.

See Also
Package.load, Data Package Specifications
**Package.load**

*Instantiate Data Package class*

**Description**

Constructor to instantiate Package class.

**Usage**

```
Package.load(descriptor = list(), basePath = NA, strict = FALSE)
```

**Arguments**

- `descriptor`  
  Data package descriptor as local path, url or object.
- `basePath`  
  Base path for all relative paths
- `strict`  
  Strict flag to alter validation behavior. Setting it to TRUE leads to throwing errors on any operation with invalid descriptor.

**See Also**

- `Package`
- `Data Package Specifications`

**Examples**

```r
# Load local descriptor
descriptor <- system.file('extdata/dp1/datapackage.json',
                           package = "datapackage.r")
dataPackage <- Package.load(descriptor)
dataPackage$descriptor

# Retrieve Package Descriptor
descriptor2 <- '{"resources": [{"name": "name", "data": "data"]}]'}
dataPackage2 <- Package.load(descriptor2)
dataPackage2$descriptor

# Expand Resource Descriptor
descriptor3 <- helpers.from.json.to.list('{"resources": [{
  "name": "name",
  "data": "data"
}]}')
dataPackage3 <- Package.load(descriptor3)
dataPackage3$descriptor
```
# Expand Tabular Resource Schema

```r
descriptor4 <- helpers.from.json.to.list("{
  "resources": [{
    "name": "name",
    "data": ["data"],
    "profile": "tabular-data-resource",
    "schema": {
      "fields": [{
        "name": "name"
      }]
    }
  }]
}")
dataPackage4 <- Package.load(descriptor4)
dataPackage4$descriptor
```

# Expand Tabular Resource Dialect

```r
descriptor5 <- helpers.from.json.to.list("{
  "resources": [{
    "name": "name",
    "data": ["data"],
    "profile": "tabular-data-resource",
    "dialect": {
      "delimiter": "custom"
    }
  }]
}")
dataPackage5 <- Package.load(descriptor5)
dataPackage5$descriptor
```

# Add, Get and Remove Package Resources

```r
descriptor6 <- helpers.from.json.to.list(  
  system.file("extdata/dp1/datapackage.json",  
  package = "datapackage.r")
)
dataPackage6 <- Package.load(descriptor6)
resource6 <- dataPackage6$addResource(  
  helpers.from.json.to.list(  
    ""name": "name",  
    "data": ["test"]  
  )
)
dataPackage6$resources[[2]]$source
# Get resource
dataPackage6$getResource('name')
# Remove resource
dataPackage6$removeResource('name')
dataPackage6$getResource('name')
```

# Modify and Commit Data Package

```r
descriptor7 <- helpers.from.json.to.list(  
'{"resources": [{"name": "name", "data": ["data"]}]}'
)
dataPackage7 <- Package.load(descriptor7)
dataPackage7$descriptor$resources[[1]]$name <- 'modified'
## Name did not modified.
dataPackage7$resources[[1]]$name
## Should commit the changes
dataPackage7$commit() # TRUE - successful commit

dataPackage7$resources[[1]]$name

---

Profile    Profile class

Description

Class to represent JSON Schema profile from Profiles Registry.

Usage

# Profile.load(profile)

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.
Methods

Public methods:
• `Profile$new()`
• `Profile$validate()`
• `Profile$clone()`

Method `new()`:

Usage:
`Profile$new(profile)`

Arguments:
- `profile` string profile name in registry or URL to JSON Schema

Method `validate()`:

Usage:
`Profile$validate(descriptor)`

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
`Profile$clone(deep = FALSE)`

Arguments:
- `deep` Whether to make a deep clone.

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
push

Push elements in a list or vector

Description
Helper function to add components in a list or vector.

Usage
push(x, value)

Arguments

x
list or vector

value
object to push in x

Resource

Resource class

Description
A class for working with data resources. You can read or iterate tabular resources using the iter/read methods and all resource as bytes using rowIter/rowRead methods.

Usage

# Resource.load(descriptor = list(), basePath = NA, strict = FALSE, dataPackage = list())

Format

R6Class object.

Value

Object of R6Class.

Methods

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

iter(keyed, extended, cast = TRUE, relations = FALSE, stream = FALSE) Only for tabular resources - 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.
Resource

- cast Disable data casting if FALSE.
- relations If TRUE foreign key fields will be checked and resolved to its references.
- stream Return Readable Stream of table rows if TRUE.

read(keyed, extended, cast = TRUE, relations = FALSE, limit) Only for tabular resources. Read the whole table and returns as list 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 If TRUE foreign key fields will be checked and resolved to its references.
- limit Integer limit of rows to return if specified.

checkRelations() Only for tabular resources. It checks foreign keys and raises an exception if there are integrity issues. Returns TRUE if no issues.

rawIter(stream = FALSE) Iterate over data chunks as bytes. If stream is TRUE Iterator will be returned.

- stream Iterator will be returned.

rawRead() Returns resource data as bytes.

infer() Infer resource metadata like name, format, mediatype, encoding, schema and profile. It commits this changes into resource instance. Returns resource descriptor.

commit(strict) Update resource instance if there are in-place changes in the descriptor. Returns TRUE on success and FALSE if not modified.

- strict Boolean - Alter strict mode for further work.

save(target) For now only descriptor will be saved. Save resource to target destination.

- target String path where to save a resource.

Properties

valid Returns validation status. It always TRUE in strict mode.
errors Returns validation errors. It always empty in strict mode.
profile Returns an instance of Profile class.
descriptor Returns list of resource descriptor.
name Returns a string of resource name.
inline Returns TRUE if resource is inline.
local Returns TRUE if resource is local.
remote Returns TRUE if resource is remote.
multipart Returns TRUE if resource is multipart.
tabular Returns TRUE if resource is tabular.
source Returns a list/string of data/path property respectively.
headers Returns a string of data source headers.
schema Returns a Schema instance to interact with data schema. Read API documentation - tableschema.Schema or Schema
Details

The Data Resource format describes a data resource such as an individual file or table. The essence of a Data Resource is a locator for the data it describes. A range of other properties can be declared to provide a richer set of metadata.

Packaged data resources are described in the resources property of the package descriptor. This property MUST be an array of objects. Each object MUST follow the Data Resource specification.

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.

Methods

Public methods:
- `Resource$new()`
- `Resource$iter()`
- `Resource$read()`
- `Resource$checkRelations()`
- `Resource$rawIter()`
- `Resource$rawRead()`
- `Resource$infer()`
- `Resource$commit()`
- `Resource$save()`
- `Resource$clone()`

**Method** `new()`:

*Usage:*

`Resource$new(descriptor, basePath, strict = FALSE, dataPackage = list())`

**Method** `iter()`:

*Usage:*

`Resource$iter(relations = FALSE, options = list())`

**Method** `read()`:

*Usage:*

`Resource$read(relations = FALSE, ...)`

**Method** `checkRelations()`:

*Usage:*

`Resource$checkRelations()`

**Method** `rawIter()`:

*Usage:*

`Resource$rawIter(stream = FALSE)`
Method rawRead():
Usage:
Resource$rawRead()

Method infer():
Usage:
Resource$infer()

Method commit():
Usage:
Resource$commit(strict = NULL)

Method save():
Usage:
Resource$save(target)

Method clone(): The objects of this class are cloneable with this method.
Usage:
Resource$clone(deep = FALSE)
Arguments:
deep Whether to make a deep clone.

See Also
Resource.load, Data Resource Specifications

| Resource.load | Instantiate Resource class |

Description
Constructor to instantiate Resource class.

Usage
Resource.load(descriptor = list(), basePath = NA, strict = FALSE, dataPackage = list())

Arguments
<table>
<thead>
<tr>
<th>descriptor</th>
<th>Data resource descriptor as local path, url or object</th>
</tr>
</thead>
<tbody>
<tr>
<td>basePath</td>
<td>Base path for all relative paths</td>
</tr>
<tr>
<td>strict</td>
<td>Strict flag to alter validation behavior. Setting it to TRUE leads to throwing errors on any operation with invalid descriptor.</td>
</tr>
<tr>
<td>dataPackage</td>
<td>data package list</td>
</tr>
</tbody>
</table>
Value

Resource class object

See Also

Resource, Data Resource Specifications

Examples

# Resource Load - with base descriptor
descriptor <-'("name": "name","data": ["data"] )' 
resource <- Resource.load(descriptor)
resource$name
resource$descriptor

# Resource Load - with tabular descriptor
descriptor2 <-'("name": "name","data": ["data"],"profile": "tabular-data-resource")' 
resource2 <- Resource.load(descriptor2)
resource2$name
resource2$descriptor

# Retrieve Resource Descriptor
descriptor3 <-'("name": "name","data": "data")'
resource3 <- Resource.load(descriptor3)
resource3$descriptor

# Expand Resource Descriptor - General Resource
descriptor4 <-'("name": "name","data": "data")'
resource4 <- Resource.load(descriptor4)
resource4$descriptor

# Expand Resource Descriptor - Tabular Resource Dialect
descriptor5 <- helpers.from.json.to.list('{
  "name": "name",
  "data": "data",
  "profile": "tabular-data-resource",
  "dialect": {"delimiter": "custom"}
} )'
resource5 <- Resource.load(descriptor5)
resource5$descriptor

# Resource - Inline source/sourceType
descriptor6 <-'("name": "name","data": "data","path": ["path"] )'
resource6 <- Resource.load(descriptor6)
resource6$source

# Resource - Remote source/sourceType
retrieveDescriptor

Retrieve descriptor

descriptor7 <- '{"name": "name","path": ["https://example.com/table.csv"]}'
resource7 <- Resource.load(descriptor7)
resource7$source

# Resource - Multipart Remote source/sourceType
descriptor8 <- '{
  "name": "name",
  "path": ["https://example.com/chunk1.csv", "https://example.com/chunk2.csv"]
}'
resource8 <- Resource.load(descriptor8)
resource8$source

# Inline Table Resource
descriptor9 <- '{
  "name": "example",
  "profile": "tabular-data-resource",
  "data": [
    ["height", "age", "name"],
    ["180", "18", "Tony"],
    ["192", "32", "Jacob"]
  ],
  "schema": {
    "fields": [
      {"name": "height", "type": "integer"},
      {"name": "age", "type": "integer"},
      {"name": "name", "type": "string"
    ]
  }
}
resource9 <- Resource.load(descriptor9)
table <- resource9$table$read()
table

describeDescriptor

Description

Helper function to retrieve descriptor.
**validate**

**Usage**

`retrieveDescriptor(descriptor)`

**Arguments**

- `descriptor` descriptor

**Description**

A standalone function to validate a data package descriptor.

**Usage**

`validate(descriptor)`

**Arguments**

- `descriptor` data package descriptor, one of:
  - string with the local CSV file (path)
  - string with the remote CSV file (url)
  - list object

**Value**

A list with:

- `valid` TRUE if valid
- `errors` a list with errors if valid FALSE

**See Also**

[https://github.com/frictionlessdata/datapackage-r#validate](https://github.com/frictionlessdata/datapackage-r#validate)

**Examples**

`validate(descriptor = '{"name": "Invalid Datapackage"}')`
validateDialect | Validate dialect

Description
Helper function to validate dialect. quoteChar and escapeChar are mutually exclusive: https://frictionlessdata.io/specs/csv-dialect/#specification

Usage
validateDialect(dialect = NULL)

Arguments
dialect | list

Value
dialect list

write.json | Save json file

Description
Save a list object in json file to disk

Usage
write.json(x, file)

Arguments
x | list object
file | file path
Index

* data
  BinaryReadable, 7
  BinaryReadableConnection, 8
  DataPackageError, 9
  Package, 20
BinaryReadable, 7
BinaryReadableConnection, 8
datapackage.r-package, 3
DataPackageError, 9
dereferencePackageDescriptor, 10
dereferenceResourceDescriptor, 10
descriptor.pointer, 11
expandPackageDescriptor, 11
expandResourceDescriptor, 11
file_basename, 12
file_extension, 12
filterDefaultDialect, 12
findFiles, 13

getwd, 17
helpers.from.json.to.list, 3, 6, 13, 21
helpers.from.list.to.json, 3, 7, 14, 21

infer, 14
is.compressed, 15
is.empty, 15
is.git, 16
is.json, 16
is.local.descriptor.path, 17
is.valid, 17
isRemotePath, 18
isSafePath, 18
isUndefined, 19

list objects, 7
locateDescriptor, 19

Package, 20, 23
Package.load, 20, 22, 23
Profile, 21, 25, 26, 28
Profile.load, 25, 26
push, 27

R6Class, 7–9, 20, 25, 27
Resource, 4, 27, 31
Resource.load, 27, 30, 30
retrieveDescriptor, 32

Schema, 28
tableschema.r::TableSchemaError, 9

validate, 33
validateDialect, 34

write.json, 34

35