Package ‘alabaster.base’

October 15, 2023

Title Save Bioconductor Objects To File

Version 1.0.0

Date 2023-04-22

License MIT + file LICENSE

Description Save Bioconductor data structures into file artifacts, and load them back into memory.
This is a more robust and portable alternative to serialization of such objects into RDS files.
Each artifact is associated with metadata for further interpretation;
downstream applications can enrich this metadata with context-specific properties.

Imports alabaster.schemas, methods, utils, S4Vectors, rhdf5, jsonlite,
jsonvalidate, Rcpp

Suggests BiocStyle, rmarkdown, knitr, testthat, digest, Matrix

LinkingTo Rcpp, Rhdf5lib

VignetteBuilder knitr

SystemRequirements C++17, GNU make

RoxygenNote 7.2.3

biocViews DataRepresentation, DataImport

git_url https://git.bioconductor.org/packages/alabaster.base

git_branch RELEASE_3_17

git_last_commit 97c0956

git_last_commit_date 2023-04-25

Date/Publication 2023-10-15

Author Aaron Lun [aut, cre]

Maintainer Aaron Lun <`infinite.monkeys.with.keyboards@gmail.com`>

R topics documented:

`.createRedirection` *Create a redirection file*

Description

Create a redirection to another path in the same staging directory. This is useful for creating shorthand aliases for resources that have inconveniently long paths.

Usage

```
.createRedirection(dir, src, dest)
```

Arguments

`dir`	String containing the path to the staging directory.
`src`	String containing the source path relative to `dir`.
`dest`	String containing the destination path relative to `dir`. This may be any path that can also be used in `acquireMetadata`.

Details

`src` should not correspond to an existing file inside `dir`. This avoids ambiguity when attempting to load `src` via `acquireMetadata`. Otherwise, it would be unclear as to whether the user wants the file at `src` or the redirection target `dest`.

`src` may correspond to existing directories. This is because directories cannot be used in `acquireMetadata`, so no such ambiguity exists.

Value

A list of metadata that can be processed by `.writeMetadata`.

Author(s)

Aaron Lun

Examples

```
# Staging an example DataFrame:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
dir.create(tmp)
info <- stageObject(df, tmp, path="coldata")
.writeMetadata(info, tmp)

# Creating a redirection:
redirect <- .createRedirection(tmp, "foobar", "coldata/simple.csv.gz")
.writeMetadata(redirect, tmp)

# We can then use this redirect to pull out metadata:
info2 <- acquireMetadata(tmp, "foobar")
str(info2)
```

`.loadObject` *Alter the loading function*

Description

Allow Artificer applications to specify an alternative loading function in `.loadObject`.

Usage

```
.loadObject(...)

.altLoadObject(load)
```

Arguments

`...`	Further arguments to pass to `loadObject` or its equivalent.
`load`	Function that can serve as a drop-in replacement for `loadObject`.

Details

`.loadObject` is just a wrapper around `loadObject` that responds to any setting of `.altLoadObject`. This allows Artificer applications to inject customizations into the loading process, e.g., to add more metadata to particular objects. Developers of Artificer extensions should use `.loadObject` to load child objects when writing their own loading functions.

To motivate the use of `.loadObject`, consider the following scenario.

1. We have created a loading function `loadX` function to load an instance of class X in an Artificer extension. This function may be called by `loadObject` if instances of X are children of other objects.
2. An Artificer application Y requires the addition of some custom metadata during the loading process for X. It defines an alternative loading function `loadObject2` that, upon encountering a schema for X, redirects to a application-specific loader `loadX2`. A typical implementation for `loadX2` would be to call `loadX` and decorate the result with the necessary metadata.
3. When operating in the context of application Y, the `loadObject2` generic is used to set `.altLoadObject`. Any calls to `.loadObject` in Y's context will subsequently call `loadObject2`.
4. So, when writing a loading function in an Artificer extension for a class that might contain X as children, we use `.loadObject` instead of directly using `loadObject`. This ensures that, if a child instance of X is encountered *and* we are operating in the context of application Y, we correctly call `loadObject2` and then ultimately `loadX2`.

For *application* developers: the alternative loading function for X should *not* call `.loadObject` on the same instance of X. Doing so will introduce an infinite recursion where `.loadObject` calls the alternative function that then calls `.loadObject`, etc. Rather, developers should either call `loadObject` or `loadX` directly. For child objects, no infinite recursion will occur and `loadObject2` or `.loadObject` can be used.

Value

For `.loadObject`, any R object similar to those returned by `loadObject`.

For `.altLoadObject`, the alternative function (if any) is returned if `load` is missing. If `load` is provided, it is used to define the alternative, and the previous alternative is returned.

Author(s)

Aaron Lun

Examples

```
old <- .altLoadObject()

# Setting it to something.
.altLoadObject(function(...) {
    print("YAY")
    loadObject(...)
})

# Staging an example DataFrame:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
dir.create(tmp)
out <- stageObject(df, tmp, path="coldata")

# And now loading it - this should print our message.
.loadObject(out, tmp)
```

```
# Restoring the old loader:
.altLoadObject(old)
```

`.processMcols` *Process R-level metadata*

Description

Stage `metadata` and `mcols` for a Annotated or Vector object, respectively. This is typically used inside `stageObject` methods for concrete subclasses.

Usage

```
.processMcols(x, dir, path, mcols.name)

.processMetadata(x, dir, path, meta.name)
```

Arguments

`x`	An Vector object for `.processMcols`, and a Annotated object for `.processMetadata`.
`dir`	String containing the path to a staging directory.
`path`	String containing the path within `dir` that is used to save all of `x`. This should be the same as that passed to the `stageObject` method.
`mcols.name`	String containing the name of the file to save `mcols(x)`. If `NULL`, no saving is performed.
`meta.name`	String containing the name of the file to save `metadata(x)`. If `NULL`, no saving is performed.

Details

If `mcols(x)` has no columns, nothing is saved by `.processMcols`. Similarly, if `metadata(x)` is an empty list, nothing is saved by `.processMetadata`.

Value

Both functions return a list containing `resource`, itself a list specifying the path within `dir` where the metadata was saved. Alternatively `NULL` if no saving is performed.

Author(s)

Aaron Lun

See Also

`.restoreMetadata`, which does the loading.

`.restoreMetadata` *Restore R-level metadata*

Description

Restore `metadata` and `mcols` for a Annotated or Vector object, respectively. This is typically used inside loading functions for concrete subclasses.

Usage

```
.restoreMetadata(x, mcol.data, meta.data, project)
```

Arguments

`x`	An Vector or Annotated object.
`mcol.data`	List of metadata specifying the `resource` location of the `mcols` files. This can also be `NULL`, in which case no `mcols` are restored.
`meta.data`	List of metadata specifying the `resource` location of the `metadata` files. This can also be `NULL`, in which case no `metadata` are restored.
`project`	Any argument accepted by the acquisition functions, see ?`acquireFile`. By default, this should be a string containing the path to a staging directory.

Value

`x` is returned, possibly with `mcols` and `metadata` added to it.

Author(s)

Aaron Lun

See Also

`.processMetadata`, which does the staging.

`.stageObject` *Alter the staging generic*

Description

Allow Artificer applications to divert to a different staging generic from `stageObject`.

Usage

```
.stageObject(...)

.altStageObject(generic)
```

Arguments

`...`	Further arguments to pass to `stageObject` or an equivalent generic.
`generic`	Generic function that can serve as a drop-in replacement for `stageObject`.

Details

`.stageObject` is just a wrapper around `stageObject` that responds to any setting of `.altStageObject`. This allows applications to inject customizations into the staging process, e.g., to store more metadata to particular objects. Developers of Artificer extensions should use `.stageObject` to stage child objects when writing their `stageObject` methods.

To motivate the use of `.stageObject`, consider the following scenario.

1. We have created a staging method for class X, defined for the `stageObject` generic.
2. An Artificer application Y requires the addition of some custom metadata during the staging process for X. It defines an alternative staging generic `stageObject2` that, upon encountering an instance of X, redirects to a application-specific method. For example, the `stageObject2` method for X could call X's `stageObject` method, add the necessary metadata to the result, and then return the list.
3. When operating in the context of application Y, the `stageObject2` generic is used to set `.altStageObject`. Any calls to `.stageObject` in Y's context will subsequently call `stageObject2`.
4. So, when writing a `stageObject` method for any objects that might include X as children, we use `.stageObject` instead of directly using `stageObject`. This ensures that, if a child instance of X is encountered *and* we are operating in the context of application Y, we correctly call `stageObject2` and then ultimately the application-specific method.

For *application* developers: the alternative `stageObject2` method for X should *not* call `.stageObject` on the same instance of X. Doing so will introduce an infinite recursion where `.stageObject` calls the alternative generic that then calls `.stageObject`, etc. Rather, developers should call `stageObject` directly in such cases. For child objects, no infinite recursion will occur and either `stageObject2` or `.stageObject` can be used.

Value

For `.stageObject`, a named list similar to the return value of `stageObject`.

For `.altStageObject`, the alternative generic (if any) is returned if `generic` is missing. If `generic` is provided, it is used to define the alternative, and the previous alternative is returned.

Author(s)

Aaron Lun

Examples

```
old <- .altStageObject()

# Creating a new generic for demonstration purposes:
setGeneric("superStageObject", function(x, dir, path, child=FALSE, ...)
    standardGeneric("superStageObject"))
```

```
setMethod("superStageObject", "ANY", function(x, dir, path, child=FALSE, ...) {
    print("Falling back to the base method!")
    stageObject(x, dir, path, child=child, ...)
})

.altStageObject(superStageObject)

# Staging an example DataFrame. This should print our message.
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
dir.create(tmp)
out <- .stageObject(df, tmp, path="coldata")

# Restoring the old loader:
.altStageObject(old)
```

`.writeMetadata` *Saving the metadata*

Description

Helper function to write metadata from a named list to a JSON file. This is commonly used inside `stageObject` methods to create the metadata file for a child object.

Usage

```
.writeMetadata(meta, dir, ignore.null = TRUE)
```

Arguments

`meta`	A named list containing metadata. This should contain at least the `"$schema"` and `"path"` elements.
`dir`	String containing a path to the staging directory.
`ignore.null`	Logical scalar indicating whether `NULL` values should be ignored during coercion to JSON.

Details

Any `NULL` values in `meta` are pruned out prior to writing when `ignore.null=TRUE`. This is done recursively so any `NULL` values in sub-lists of `meta` are also ignored.

Any scalars are automatically unboxed so array values should be explicitly specified as such with `I()`.

Any starting `"./"` in `meta$path` will be automatically removed. This allows staging methods to save in the current directory by setting `path="."`, without the need to pollute the paths with a `"./"` prefix.

The JSON-formatted metadata is validated against the schema in `meta[["$schema"]]` using **json-validate**. The location of the schema is taken from the `package` attribute in that string, if one exists; otherwise, it is assumed to be in the **alabaster.schemas** package. (All schemas are assumed to live in the `inst/schemas` subdirectory of their indicated packages.)

We also use the schema to determine whether `meta` refers to an actual artifact or is a metadata-only document. If it refers to an actual file, we compute its MD5 sum and store it in the metadata for saving. We also save its associated metadata into a JSON file at a location obtained by appending `".json"` to `meta$path`.

For artifacts, the MD5 sum calculation will be skipped if the `meta` already contains a `md5sum` field. This can be useful on some occasions, e.g., to improve efficiency when the MD5 sum was already computed during staging, or if the artifact does not actually exist in its full form on the file system.

Value

A JSON file containing the metadata is created at `path`. A list of resource metadata is returned, e.g., for inclusion as the `"resource"` property in parent schemas.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
dir.create(tmp)
info <- stageObject(df, tmp, path="coldata")
.writeMetadata(info, tmp)
cat(readLines(file.path(tmp, "coldata/simple.csv.gz.json")), sep="\n")
```

`acquireFile` *Acquire file or metadata*

Description

Acquire a file or metadata for loading. As one might expect, these are typically used inside a `load*` function.

Usage

```
acquireFile(project, path)

acquireMetadata(project, path)

## S4 method for signature 'character'
```

```
acquireFile(project, path)

## S4 method for signature 'character'
acquireMetadata(project, path)
```

Arguments

`project`	Any value specifying the project of interest. The default methods expect a string containing a path to a staging directory, but other objects can be used to control dispatch.
`path`	String containing a relative path to a resource inside the staging directory.

Details

By default, files and metadata are loaded from the same staging directory that is written to by `stageObject`. Artificer applications can define custom methods to obtain the files and metadata from a different location, e.g., remote databases. This is achieved by dispatching on a different value of `project`.

Each custom acquisition method should take two arguments. The first argument is an R object representing some concept of a “project”. In the default case, this is a string containing a path to the staging directory representing the project. However, it can be anything, e.g., a number containing a database identifier, a list of identifiers and versions, and so on - as long as the custom acquisition method is capable of understanding it, the `load*` functions don’t care. The second argument is a string containing the relative path to the resource inside that project.

The return value for each custom acquisition function should be the same as their local counterparts. That is, any custom file acquisition function should return a file path, and any custom metadata acquisition function should return a naamed list of metadata.

Value

`acquireFile` methods return a local path to the file corresponding to the requested resource.

`acquireMetadata` methods return a named list of metadata for the requested resource.

Author(s)

Aaron Lun

Examples

```
# Staging an example DataFrame:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
tmp <- tempfile()
dir.create(tmp)
info <- stageObject(df, tmp, path="coldata")
.writeMetadata(info, tmp)

# Retrieving the metadata:
meta <- acquireMetadata(tmp, "coldata/simple.csv.gz")
```

```
str(meta)

# Retrieving the file:
acquireFile(tmp, "coldata/simple.csv.gz")
```

`checkValidDirectory` *Check if a staging directory is valid*

Description

Check whether a staging directory is valid in terms of its structure and metadata.

Usage

```
checkValidDirectory(dir)
```

Arguments

`dir`	String containing the path to a staging directory.

Details

This function verifies that the restrictions described in `stageObject` are respected, namely:

- Each object is represented by subdirectory with a single JSON document.
- Each JSON metadata document's `path` property exists and is consistent with the path of the document itself.
- Child objects are nested in subdirectories of the parent object's directory.
- Child objects have the `is_child` property set to true in their metadata.
- Each child object is referenced exactly once in its parent object's metadata.

This function will also check that redirections are valid:

- The `path` property of the redirection does *not* exist and is consistent with the path of the redirection document.
- The redirection target location exists in the directory.

Value

`NULL` invisibly on success, otherwise an error is raised.

Author(s)

Aaron Lun

Examples

```
# Mocking up an object:
library(S4Vectors)
ncols <- 123
df <- DataFrame(
    X = rep(LETTERS[1:3], length.out=ncols),
    Y = runif(ncols)
)
df$Z <- DataFrame(AA = sample(ncols))

# Mocking up the directory:
tmp <- tempfile()
dir.create(tmp, recursive=TRUE)
.writeMetadata(stageObject(df, tmp, "foo"), tmp)

# Checking that it's valid:
checkValidDirectory(tmp)
```

`loadBaseList` *Load a base list*

Description

Load a list from file, possibly containing complex entries.

Usage

```
loadBaseList(info, project, parallel = TRUE)
```

Arguments

`info`	Named list containing the metadata for this object.
`project`	Any argument accepted by the acquisition functions, see ?`acquireFile`. By default, this should be a string containing the path to a staging directory.
`parallel`	Whether to perform reading and parsing in parallel for greater speed. Only relevant for lists stored in the JSON format.

Details

This function effectively reverses the behavior of "`stageObject,list-method`", loading the list back into memory from the JSON file. Atomic vectors, arrays and data frames are loaded directly while complex values are loaded by calling the appropriate loading function.

Value

The list described by `info`.

Author(s)

Aaron Lun

See Also

"stageObject,list-method", for the staging method.

Examples

```
library(S4Vectors)
ll <- list(A=1, B=LETTERS, C=DataFrame(X=letters))

# First staging it:
tmp <- tempfile()
dir.create(tmp)
info <- stageObject(ll, tmp, path="stuff")

# And now loading it:
loadBaseList(info, tmp)
```

---

loadDataFrame *Load a DataFrame*

Description

Load a DataFrame from file, possibly containing complex columns and row names.

Usage

```
loadDataFrame(info, project, include.nested = TRUE, parallel = TRUE)
```

Arguments

`info`	Named list containing the metadata for this object.
`project`	Any argument accepted by the acquisition functions, see ?`acquireFile`. By default, this should be a string containing the path to a staging directory.
`include.nested`	Logical scalar indicating whether nested DataFrames should be loaded.
`parallel`	Whether to perform reading and parsing in parallel for greater speed.

Details

This function effectively reverses the behavior of "stageObject,DataFrame-method", loading the DataFrame back into memory from the CSV or HDF5 file. Atomic columns are loaded directly while complex columns (such as nested DataFrames) are loaded by calling the appropriate `restore` method.

One implicit interpretation of using a nested DataFrame is that the contents are not important enough to warrant top-level columns. In such cases, we can skip all columns containing a nested DataFrame by setting `include.nested=FALSE`. This avoids the cost of loading a (potentially large) nested DataFrame when its contents are unlikely to be relevant.

Value

The DataFrame described by `info`.

Author(s)

Aaron Lun

See Also

`"stageObject,DataFrame-method"`, for the staging method.

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

# First staging it:
tmp <- tempfile()
dir.create(tmp)
out <- stageObject(df, tmp, path="coldata")

# And now loading it:
loadDataFrame(out, tmp)
```

`loadDataFrameFactor` *Load an DataFrame factor*

Description

Load a Factor where the levels are rows of a DataFrame.

Usage

```
loadDataFrameFactor(info, project)
```

Arguments

`info`	Named list containing the metadata for this object.
`project`	Any argument accepted by the acquisition functions, see `?acquireFile`. By default, this should be a string containing the path to a staging directory.

Value

A DataFrameFactor described by `info`.

Author(s)

Aaron Lun

See Also

`"stageObject,DataFrameFactor-method"`, for the staging method.

Examples

```
library(S4Vectors)
df <- DataFrame(X=LETTERS[1:5], Y=1:5)
out <- DataFrameFactor(df[sample(5, 100, replace=TRUE),,drop=FALSE])

# Staging the object:
tmp <- tempfile()
dir.create(tmp)
info <- stageObject(out, tmp, path="test")

# Loading it back in:
loadDataFrameFactor(info, tmp)
```

`loadObject` *Load an object from its metadata*

Description

Load an object from its metadata, based on the loading function specified in its schema.

Usage

```
loadObject(info, project, ...)
```

Arguments

`info`	Named list containing the metadata for this object.
`project`	Any argument accepted by the acquisition functions, see ?`acquireFile`. By default, this should be a string containing the path to a staging directory.
`...`	Further arguments to pass to the specific loading function listed in the schema.

Details

The `loadObject` function is intended to load objects where the class is not known in advance. Most typically, this is indirectly called inside other loading functions to restore *child* objects of arbitrary type. Once in memory, the child objects can then be assembled into more complex objects by the caller. (It would be unwise to use `loadObject` to try restore a non-child object as this would result in infinite recursion.)

This function will look through the schemas in **alabaster.schemas** to find the schema specified in `` info$`$schema` ``. Upon discovery, `loadObject` will extract the loading function from the `_attributes.restore.R` property of the schema; this should be a string that contains a namespaced function, which can be parsed and evaluated to obtain said function. `loadObject` will then call the loading function with the supplied arguments. Developers can temporarily add extra packages to the schema search path by supplying package names in the `alabaster.schema.locations` option; schema files are expected to be stored in the `schemas` subdirectory of each package's installation directory.

Developers of Artificer extensions should use `.loadObject` rather than calling `loadObject` directly. This ensures that any application-level overrides of the loading functions are respected. Developers of Artificer applications should also read the commentary in `?".altLoadObject"`.

Value

An object corresponding to `info`, as defined by the loading function.

Author(s)

Aaron Lun

Examples

```
# Same example as stageObject, but reversed.
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

# First staging it:
tmp <- tempfile()
dir.create(tmp)
out <- stageObject(df, tmp, path="coldata")

# Loading it:
loadObject(out, tmp)
```

`quickReadCsv` *Quickly read and write a CSV file*

Description

Quickly read and write a CSV file, usually as a part of staging or loading a larger object. This assumes that all files follow the comservatory specification.

Usage

```
.quickReadCsv(
  path,
  expected.columns,
  expected.nrows,
  compression,
  row.names,
  parallel = TRUE
)

.quickWriteCsv(df, path, ..., row.names = FALSE, compression = "gzip")
```

Arguments

`path`	String containing a path to a CSV to read/write.
`expected.columns`	Named character vector specifying the type of each column in the CSV (excluding the first column containing row names, if `row.names=TRUE`).
`expected.nrows`	Integer scalar specifying the expected number of rows in the CSV.
`compression`	String specifying the compression that was/will be used. This should be either `"none"`, `"gzip"`.
`row.names`	For `.quickReadCsv`, a logical scalar indicating whether the CSV contains row names.
`parallel`	Whether reading and parsing should be performed concurrently. For `.quickWriteCsv`, a logical scalar indicating whether to save the row names of `df`.
`df`	A DataFrame or data.frame object, containing only atomic columns.
`...`	Further arguments to pass to `write.csv`.

Value

For `.quickReadCsv`, a DataFrame containing the contents of `path`.

For `.quickWriteCsv`, `df` is written to `path` and a `NULL` is invisibly returned.

Author(s)

Aaron Lun

Examples

```
library(S4Vectors)
df <- DataFrame(A=1, B="Aaron")

temp <- tempfile()
.quickWriteCsv(df, path=temp, row.names=FALSE, compression="gzip")

.quickReadCsv(temp, c(A="numeric", B="character"), 1, "gzip", FALSE)
```

`readLocalObject` *Convenience helpers for handling local directories*

Description

Read, write and list objects from a local staging directory. These are just convenience wrappers around functions like `loadObject`, `stageObject` and `.writeMetadata`.

Usage

```
readLocalObject(dir, path, ...)

saveLocalObject(x, dir, path, ...)

listLocalObjects(dir, includeChildren = FALSE)
```

Arguments

`dir`	String containing a path to the directory.
`path`	String containing a relative path to the object of interest inside `dir`.
`...`	Further arguments to pass to `loadObject` (for `readLocalObject`) or `stageObject` (for `saveLocalObject`).
`x`	Object to be saved.
`includeChildren`	Logical scalar indicating whether child objects should be returned.

Value

For `readLocalObject`, the object at `path`.

For `saveLocalObject`, the object is saved to `path` inside `dir`. All necessary directories are created if they are not already present. A `NULL` is returned invisibly.

For `listLocalObjects`, a named list where each entry corresponds to an object and is named after the path to that object inside `dir`. The value of each element is another list containing the metadata for its corresponding object.

Author(s)

Aaron Lun

Examples

```
local <- tempfile()

# Creating a slightly complicated object:
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])
```

```
df$C <- DataFrame(D=letters[1:10], E=runif(10))

# Saving it:
saveLocalObject(df, local, "FOOBAR")

# Reading it back:
readLocalObject(local, "FOOBAR")

# Listing all available objects:
names(listLocalObjects(local))
names(listLocalObjects(local, includeChildren=TRUE))
```

---

saveFormats *Choose the format for certain objects*

Description

Alter the format used to save DataFrames or base lists in their respective `stageObject` methods.

Usage

```
.saveDataFrameFormat(format)

.saveBaseListFormat(format)
```

Arguments

`format`	String containing the format to use. • For `saveDataFrameFormat`, this may be `"csv"`, `"csv.gz"` (default) or `"hdf5"`. • For `saveBaseListFormat`, this may be `"json.gz"` (default) or `"hdf5"`. Alternatively `NULL`, to use the default format.

Details

`stageObject` methods will treat a `format=NULL` in the same manner as the default format. The distinction exists to allow downstream applications to set their own defaults while still responding to user specification. For example, an application can detect if the existing format is `NULL`, and if so, apply another default via `.saveDataFrameFormat`. On the other hand, if the format is not `NULL`, this is presumably specified by the user explicitly and should be respected by the application.

Value

If `format` is missing, a string containing the current format is returned, or `NULL` to use the default format.

If `format` is supplied, it is used to define the current format, and the *previous* format is returned.

Author(s)

Aaron Lun

Examples

```
(old <- .saveDataFrameFormat())

.saveDataFrameFormat("hdf5")
.saveDataFrameFormat()

# Setting it back.
.saveDataFrameFormat(old)
```

`stageObject` *Stage assorted objects*

Description

Generic to stage assorted R objects. More methods may be defined by other packages to extend the **alabaster.base** framework to new classes.

Usage

```
stageObject(x, dir, path, child = FALSE, ...)
```

Arguments

`x`	A Bioconductor object of the specified class.
`dir`	String containing the path to the staging directory.
`path`	String containing a prefix of the relative path inside `dir` where `x` is to be saved. The actual path used to save `x` may include additional components, see Details.
`child`	Logical scalar indicating whether `x` is a child of a larger object.
`...`	Further arguments to pass to specific methods.

Details

Methods for the `stageObject` generic are expected to create a subdirectory at the input `path` inside `dir`. All files (artifacts and metadata documents) required to represent `x` on disk should be created inside `path`. The expected contents of `path` are:

- Exactly one JSON metadata document with a name ending in `.json`. This contains the metadata describing `x`, according to the schema defined in the `$schema` property. If a data file exists (see next), the `path` property should contain the relative path to that file from `dir`; otherwise it should contain the relative path of the metadata document itself.

- (Optional) A file containing the data inside `x`. This file should have the same name as the metadata file after stripping the `.json` extension. Methods are free to choose any format and name within `path` except for the `.json` file extension, which is reserved for metadata documents created with `.writeMetadata`.
- (Optional) Further subdirectories containing child objects of `x`. Each child object should be saved in its own subdirectory, and should be referenced via a `resource` object within the metadata for `x`. When saving children, methods should call `.stageObject` rather than `stageObject` (note the period at the start of the former). This ensures that the staging method will respect customizations from alabaster applications that define their own generic in `.altStageObject`.

Methods can create both a data file and multiple subdirectories. In this manner, we can decompose complex `x` into their components for easier handling.

Keep in mind that the returned `path` will differ from the input `path`; the latter refers to the directory while the former refers to a file inside the directory. Methods should use the former to reference `x` from the metadata of a parent object.

The `stageObject` generic will check if the `path` already exists before dispatching to the methods. If so, it will throw an error to ensure that downstream name clashes do not occur. The exception is if `path = "."`, in which case no check is performed; this is useful for eliminating subdirectories in situations where the project contains only one object.

Any attempt to use the `stageObject` generic to save another non-child object into `path` or its subdirectories will cause an error. This ensures that `path` contains all and only the contents of `x`.

Value

`dir` is populated with files containing the contents of `x`. All created files should be prefixed by `path`, either in the file name or as part of a subdirectory path.

A named list containing the metadata for `x` is returned, containing at least:

- `$schema`, a string specifying the schema to use to validate the metadata. This may be decorated with the `package` attribute to help `.writeMetadata` find the package containing the schema.
- `path`, a string containing the path to some file prefixed by the input `path`.
- `is_child`, a logical scalar equal to the input `child`.

All files created by a `stageObject` method should be referenced from the metadata list, directly or otherwise (e.g., via child resources).

Author(s)

Aaron Lun

See Also

`checkValidDirectory`, for validation of the staged contents.

Examples

```
tmp <- tempfile()
dir.create(tmp)

library(S4Vectors)
X <- DataFrame(X=LETTERS, Y=sample(3, 26, replace=TRUE))
stageObject(X, tmp, path="test1")
list.files(file.path(tmp, "test1"))
```

---

stageObject,DataFrame-method
Stage a DataFrame

Description

Stage a DataFrame by saving it to a CSV or HDF5 file. CSV files follow the comservatory specification, while the expected layout of a HDF5 file is described in the `hdf5_data_frame` schema in **alabaster.schemas**.

Usage

```
## S4 method for signature 'DataFrame'
stageObject(
  x,
  dir,
  path,
  child = FALSE,
  df.name = "simple",
  mcols.name = "mcols",
  meta.name = "other"
)
```

Arguments

`x`	A DataFrame.
`dir`	String containing the path to the staging directory.
`path`	String containing a prefix of the relative path inside `dir` where `x` is to be saved. The actual path used to save `x` may include additional components, see Details.
`child`	Logical scalar indicating whether `x` is a child of a larger object.
`df.name`	String containing the relative path inside `dir` to save the CSV/HDF5 file.
`mcols.name`	String specifying the name of the directory inside `path` to save `mcols(x)`. If `NULL`, per-element metadata is not saved.
`meta.name`	String specifying the name of the directory inside `path` to save `metadata(x)`. If `NULL`, object metadata is not saved.

Details

All atomic vector types are supported in the columns along with dates and (ordered) factors. Dates and factors are converted to character vectors and saved as such inside the file. Factor levels are saved in a separate data frame, which is referenced in the `columns` field of the returned metadata.

Any non-atomic columns are saved to a separate file inside `path` via `stageObject`, and referenced from the corresponding `columns` entry. For consistency, they will be replaced in the main file by a placeholder all-zero column.

As a DataFrame is a Vector subclass, its R-level metadata can be staged by `.processMetadata`.

Value

A named list containing the metadata for `x`. `x` itself is written to a CSV or HDF5 file inside `path`. Additional files may also be created inside `path` and referenced from the metadata.

File formats

If `.saveDataFrameFormat() == "csv"`, the contents of `x` are saved to a uncompressed CSV file. If `x` has non-`NULL` row names, the first saved column in the CSV is named `row_names` and will contain the row names. This should be ignored when indexing columns and comparing them to the corresponding entry of `columns` in the file's JSON metadata document.

If `.saveDataFrameFormat() == "csv.gz"`, the CSV file is compressed (the default). This reduces space and bandwidth requirements at the cost of the (de)compression overhead. It also makes it more difficult to do queries inside the file without decompression of the entire file.

If `.saveDataFrameFormat() == "hdf5"`, `x` is saved into a HDF5 file instead of a CSV. Columns are saved into a `data` group where each column is a dataset named after its positional index. The names of the columns are saved into the `column_names` dataset. If row names are present, a separate `row_names` dataset containing the row names will be generated. This format is most useful for random access and for preserving the precision of numerical data.

Author(s)

Aaron Lun

See Also

`https://github.com/LTLA/comservatory`, for the CSV file specification.

The `csv_data_frame` and `hdf5_data_frame` schemas from the **alabaster.schemas** package.

Examples

```
library(S4Vectors)
df <- DataFrame(A=1:10, B=LETTERS[1:10])

tmp <- tempfile()
dir.create(tmp)
stageObject(df, tmp, path="coldata")

list.files(tmp, recursive=TRUE)
```

stageObject,DataFrameFactor-method

Stage a DataFrameFactor object

Description

Stage a DataFrameFactor object, a generalization of the base factor for DataFrame levels.

Usage

```
## S4 method for signature 'DataFrameFactor'
stageObject(
  x,
  dir,
  path,
  child = FALSE,
  index.name = "index",
  level.name = "levels",
  mcols.name = "mcols"
)
```

Arguments

`x`	A DataFrameFactor object.
`dir`	String containing the path to the staging directory.
`path`	String containing a prefix of the relative path inside `dir` where `x` is to be saved. The actual path used to save `x` may include additional components, see Details.
`child`	Logical scalar indicating whether `x` is a child of a larger object.
`index.name`	String containing the name of the file to save the factor indices.
`level.name`	String containing the name of the subdirectory to save the factor levels.
`mcols.name`	String specifying the name of the directory inside `path` to save the `mcols`. If `NULL`, the metadata columns are not saved.

Details

We create one file in `path` for the levels and another file for the factor indices. The levels file contains a DataFrame as staged by `stageObject,DataFrame-method`. Indices are 1-based and reference one record of the levels file. As the DataFrameFactor is a Vector subclass, its R-level metadata can be staged by `.processMetadata`.

Value

A named list containing the metadata for `x`. `x` itself is written to a file inside `path`. Additional files may also be created inside `path` and referenced from the metadata.

Author(s)

Aaron Lun

See Also

The `data_frame_factor` schema from **alabaster.schemas**.

Examples

```
library(S4Vectors)
df <- DataFrame(X=LETTERS[1:5], Y=1:5)
out <- DataFrameFactor(df[sample(5, 100, replace=TRUE),,drop=FALSE])

tmp <- tempfile()
dir.create(tmp)
stageObject(out, tmp, path="test")

list.files(file.path(tmp, "test"), recursive=TRUE)
```

`stageObject,list-method`

Stage a base list

Description

Save a list or List to a JSON or HDF5 file, with external subdirectories created for any of the more complex list elements (e.g., DataFrames, arrays). This uses the uzuki2 specification to ensure that appropriate types are declared.

Usage

```
## S4 method for signature 'list'
stageObject(x, dir, path, child = FALSE, fname = "list")

## S4 method for signature 'List'
stageObject(x, dir, path, child = FALSE, fname = "list")
```

Arguments

`x`	A Bioconductor object of the specified class.
`dir`	String containing the path to the staging directory.
`path`	String containing a prefix of the relative path inside `dir` where `x` is to be saved. The actual path used to save `x` may include additional components, see Details.
`child`	Logical scalar indicating whether `x` is a child of a larger object.
`fname`	String containing the name of the file to use to save `x`. Note that this should not have a `.json` suffix, so as to avoid confusion with the JSON-formatted metadata.

Value

A named list containing the metadata for x, where x itself is written to a JSON file.

File formats

If .saveBaseListFormat() == "json.gz", the list is saved to a Gzip-compressed JSON file (the default). This is an easily parsed format with low storage overhead.

If .saveBaseListFormat() == "hdf5", x is saved into a HDF5 file instead. This format is most useful for random access and for preserving the precision of numerical data.

Author(s)

Aaron Lun

See Also

https://github.com/LTLA/uzuki2 for the specification.

The json_simple_list and hdf5_simple_list schemas from the **alabaster.schemas** package.

Examples

```
library(S4Vectors)
ll <- list(A=1, B=LETTERS, C=DataFrame(X=1:5))

tmp <- tempfile()
dir.create(tmp)
stageObject(ll, tmp, path="stuff")

list.files(tmp, recursive=TRUE)
```

Index