Package 'biodb'

October 14, 2021

Title biodb, a library and a development framework for connecting to chemical and biological databases

Version 1.0.4

Description The biodb package provides access to standard remote chemical and biological databases (ChEBI, KEGG, HMDB, ...), as well as to in-house local database files (CSV, SQLite), with easy retrieval of entries, access to web services, search of compounds by mass and/or name, and mass spectra matching for LCMS and MSMS. Its architecture as a development framework facilitates the development of new database connectors for local projects or inside separate published packages.

biocViews Software, Infrastructure, DataImport, KEGG

Depends R (>= 4.0)

License AGPL-3

Encoding UTF-8

VignetteBuilder knitr

Suggests BiocStyle, roxygen2, devtools, testthat (>= 2.0.0), knitr, rmarkdown, covr, xml2, git2r

Imports R6, methods, chk, lgr, progress, lifecycle, XML, stringr, plyr, yaml, jsonlite, RCurl, Rcpp, rappdirs, stats, openssl, RSQLite, withr

LinkingTo Rcpp, testthat

NeedsCompilation yes

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.1

Collate 'BiodbObject.R' 'BiodbChildObject.R' 'BiodbConnBase.R' 'BiodbConn.R' 'BiodbCompounddbConn.R' 'BiodbObserver.R' 'BiodbConfig.R' 'BiodbConnObserver.R' 'BiodbEntry.R' 'BiodbCsvEntry.R' 'BiodbDbInfo.R' 'BiodbDbsInfo.R' 'BiodbDownloadable.R' 'BiodbEditable.R' 'BiodbEntryField.R' 'BiodbMain.R' 'BiodbEntryFields.R' 'BiodbFactory.R' 'BiodbXmlEntry.R' 'BiodbHtmlEntry.R' 'BiodbJsonEntry.R' 'BiodbListEntry.R' 'BiodbMassdbConn.R' 'BiodbPersitentCache.R' 'BiodbRequestScheduler.R' 'BiodbRemotedbConn.R' 'BiodbUrl.R'

'BiodbRequest.R' 'BiodbRequestSchedulerRule.R'
'BiodbTxtEntry.R' 'BiodbSdfEntry.R' 'BiodbSqlExpr.R'
'BiodbSqlBinaryOp.R' 'BiodbSqlField.R' 'BiodbSqlList.R'
'BiodbSqlLogicalOp.R' 'BiodbSqlQuery.R' 'BiodbSqlValue.R'
'BiodbWritable.R' 'CsvFileConn.R' 'CompCsvFileConn.R'
'CompCsvFileEntry.R' 'SqliteConn.R' 'CompSqliteConn.R'
'CompSqliteEntry.R' 'ExtGenerator.R' 'ExtFileGenerator.R'
'ExtConnClass.R' 'ExtCpp.R' 'ExtDefinitions.R'
'ExtDescriptionFile.R' 'ExtEntryClass.R' 'ExtGitignore.R'
'ExtLicense.R' 'ExtMakefile.R' 'ExtPackage.R'
'ExtPackageFile.R' 'ExtRbuildignore.R' 'ExtReadme.R'
'ExtTests.R' 'ExtTravisFile.R' 'ExtVignette.R' 'FileTemplate.R'
'MassCsvFileConn.R' 'MassCsvFileEntry.R' 'MassSqliteConn.R'
'MassSqliteEntry.R' 'Progress.R' 'Range.R' 'RcppExports.R'
'catch-routine-registration.R' 'fcts_biodb.R' 'fcts_ext.R'
'fcts_mass.R' 'fcts_misc.R' 'generic_tests.R' 'http_consts.R'
'package.R' 'spec-dist.R' 'test_framework.R'

git_url https://git.bioconductor.org/packages/biodb

git_branch RELEASE_3_13

git_last_commit 84b7aff

git_last_commit_date 2021-06-09

Date/Publication 2021-10-14

Author Pierrick Roger [aut, cre],
Alexis Delabrière [ctb]

Maintainer Pierrick Roger <pierrick.roger@cea.fr>

R topics documented:

`biodb-package` *biodb: biodb, a library and a development framework for connecting to chemical and biological databases*

Description

The biodb package provides access to standard remote chemical and biological databases (ChEBI, KEGG, HMDB, ...), as well as to in-house local database files (CSV, SQLite), with easy retrieval of entries, access to web services, search of compounds by mass and/or name, and mass spectra matching for LCMS and MSMS. Its architecture as a development framework facilitates the development of new database connectors for local projects or inside separate published packages.

Details

To get a presentation of the *biodb* package and get started with it, please see the "biodb" vignette.

```
vignette('biodb', package='biodb')
```

Author(s)

Maintainer: Pierrick Roger <pierrick.roger@cea.fr>

Other contributors:

- Alexis Delabrière <delabriere@imsb.biol.ethz.ch> [contributor]

See Also

BiodbMain, BiodbConfig, BiodbFactory, BiodbPersistentCache, BiodbDbsInfo, BiodbEntryFields.

BiodbCompounddbConn-class

An interface for all compound databases.

Description

This interface is inherited by all compound databases. It declares method headers specific to compound databases.

Methods

`annotateMzValues( x, mz.tol, ms.mode, mz.tol.unit = c("plain", "ppm"), mass.field = "monoisotopic.mass",`
:

Annotates a mass spectrum with the database. For each matching entry the entry field values will be set inside columns appended to the data frame. Names of these columns will use a common prefix in order to distinguish them from other data from the input data frame.

x: Either a data frame or a numeric vector containing the M/Z values.

mz.col: The name of the column where to find M/Z values in case x is a data frame.

ms.mode: The MS mode. Set it to either 'neg' or 'pos'.

mz.tol: The tolerance on the M/Z values.

mz.tol.unit: The type of the M/Z tolerance. Set it to either to 'ppm' or 'plain'.

mass.field: The mass field to use for matching M/Z values. One of: 'monoisotopic.mass', 'molecular.mass', 'average.mass', 'nominal.mass'.

fields: A character vector containing the additional entry fields you would like to get for each matched entry. Each field will be output in a different column.

insert.input.values: Insert input values at the beginning of the result data frame.

prefix: A prefix that will be inserted before the name of each added column in the output. By default it will be set to the name of the database followed by a dot.

fieldsLimit: The maximum of values to output for fields with multiple values. Set it to 0 to get all values.

max.results: If set, it is used to limit the number of matches found for each M/Z value. To get all the matches, set this parameter to NA_integer_. Default value is 3.

Returned value: A data frame containing the input values, and annotation columns appended at the end. The first annotation column contains the IDs of the matched entries. The following columns contain the fields you have requested through the 'fields' parameter.

`searchCompound( name = NULL, mass = NULL, mass.field = NULL, mass.tol = 0.01, mass.tol.unit = "plain", max.`
:

This method is deprecated. Use searchForEntries() instead.

Searches for compounds by name and/or by mass. At least one of name or mass must be set.

name: The name of a compound to search for.

description: A character vector of words or expressions to search for inside description field. The words will be searched in order. A match will be made only if all words are inside the description field.

mass: The searched mass.

mass.field: For searching by mass, you must indicate a mass field to use ('monoisotopic.mass', 'molecular.mass', 'average.mass' or 'nominal.mass').

mass.tol: The tolerance value on the molecular mass.

mass.tol.unit: The type of mass tolerance. Either 'plain' or 'ppm'.

max.results: The maximum number of matches to return.

Returned value: A character vector of entry IDs.

See Also

Super class BiodbConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a compound CSV file database
chebi.tsv <- system.file("extdata", "chebi_extract.tsv", package='biodb')

# Get the connector of a compound database
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi.tsv)

# Search for compounds
ids <- conn$searchCompound(name='prion protein', max.results=10)

# Terminate instance.
mybiodb$terminate()
```

`BiodbConfig-class` *A class for storing configuration values.*

Description

This class is responsible for storing configuration. You must go through the single instance of this class to create and set and get configuration values. To get the single instance of this class, call the `getConfig()` method of class `BiodbMain`.

Methods

`define(def)` Defines config properties from a structured object, normally loaded from a YAML file.

def: The list of key definitions.

Returned value: None.

`disable(key)` :

Set a boolean key to FALSE.

key: The name of a configuration key.

Returned value: None.

`enable(key)` :

Set a boolean key to TRUE.

key: The name of a configuration key.

Returned value: None.

`get(key)` :

Get the value of a key.

key: The name of a configuration key.

Returned value: The value associated with the key.

`getAssocEnvVar(key)` :

Returns the environment variable associated with this configuration key.

key: The name of a configuration key.

Returned value: None.

`getDefaultValue(key, as.chr = FALSE)` :

Get the default value of a key.

key: The name of a configuration key.

as.chr: If set to TRUE, returns the value as character.

Returned value: The default value for that key.

`getDescription(key)` :

Get the description of a key.

key: The name of a configuration key.

Returned value: The description of the key as a character value.

`getKeys(deprecated = FALSE)` :

Get the list of available keys.

deprecated: If set to TRUE returns also the deprecated keys.

Returned value: A character vector containing the config key names.

`getTitle(key)` :

Get the title of a key.

key: The name of a configuration key.

Returned value: The title of the key as a character value.

`hasKey(key)` :

Test if a key exists.

key: The name of a configuration key.

Returned value: TRUE if a key with this name exists, FALSE otherwise.

`isDefined(key, fail = TRUE)` :

Test if a key is defined (i.e.: if a value exists for this key).

key: The name of a configuration key.

fail: If set to TRUE and the configuration key does not exist, then an error will be raised.

Returned value: TRUE if the key has a value, FALSE otherwise.

`isEnabled(key)` :

Test if a boolean key is set to TRUE. This method will raise an error if the key is not a boolean key.

key: The name of a configuration key.

Returned value: TRUE if the boolean key has a value set to TRUE, FALSE otherwise.

`listKeys()` :

Get the full list of keys as a data frame.

Returned value: A data frame containing keys, titles, types, and default values.

`reset(key = NULL)` :

Reset the value of a key.

key: The name of a configuration key. If NULL, all keys will be reset.

Returned value: None.

`set(key, value)` :

Set the value of a key.

key: The name of a configuration key.

value: A value to associate with the key.

Returned value: None.

See Also

`BiodbMain`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get the config instance:
config <- mybiodb$getConfig()

# Print all available keys
config$getKeys()

# Get a configuration value:
value <- config$get('cache.directory')

# Set a configuration value:
config$set('dwnld.timeout', 600)

# For boolean values, you can use boolean methods:
config$get('offline')
config$enable('offline')    # set to TRUE
config$disable('offline')   # set to FALSE
config$isEnabled('offline')

# Terminate instance.
mybiodb$terminate()
```

BiodbConn-class *The mother abstract class of all database connectors.*

Description

This is the super class of all connector classes. All methods defined here are thus common to all connector classes. Some connector classes inherit directly from this abstract class. Some others inherit from intermediate classes BiodbRemotedbConn and BiodbMassdbConn. As for all connector concrete classes, you won't have to create an instance of this class directly, but you will instead go through the factory class. However, if you plan to develop a new connector, you will have to call the constructor of this class. See section Fields for a list of the constructor's parameters. Concrete classes may have direct web services methods or other specific methods implemented, in which case they will be described inside the documentation of the concrete class. Please refer to the documentation of each concrete class for more information. The database direct web services methods will be named "ws.*".

Details

The constructor has the following arguments:

id: The identifier of the connector.

cache.id: The identifier used in the disk cache.

Methods

`checkDb()` :

Checks that the database is correct by trying to retrieve all its entries.

Returned values: None.

`correctIds(ids)` :

Correct a vector of IDs by formatting them to the database official format, if required and possible.

ids: A character vector of IDs.

Returned values: The vector of IDs corrected.

`deleteAllCacheEntries()` :

Delete all entries from the memory cache. This method is deprecated, please use deleteAllEntriesFromVolatileCache() instead.

Returned value: None.

`deleteAllEntriesFromPersistentCache(deleteVolatile = TRUE)` :

Delete all entries from the persistent cache (disk cache).

deleteVolatile: If TRUE deletes also all entries from the volatile cache (memory cache).

Returned value: None.

`deleteAllEntriesFromVolatileCache()` :

Delete all entries from the volatile cache (memory cache).

Returned value: None.

`deleteWholePersistentCache(deleteVolatile = TRUE)` :

Delete all files associated with this connector from the persistent cache (disk cache). deleteVolatile: If TRUE deletes also all entries from the volatile cache (memory cache).

Returned value: None.

`getAllCacheEntries()` :

This method is deprecated.

Use getAllVolatileCacheEntries() instead.

`getAllVolatileCacheEntries()` :

Get all entries stored in the memory cache (volatile cache).

Returned value: A list of BiodbEntry instances.

`getCacheFile(entry.id)` :

Get the path to the persistent cache file.

entry.id: The identifiers (e.g.: accession numbers) as a character vector of the database entries.

Returned value: A character vector, the same length as the vector of IDs, containing the paths to the cache files corresponding to the requested entry IDs.

`getCacheId()` :

Gets the ID used by this connector in the disk cache.

Returned value: The cache ID of this connector.

`getEntry(id, drop = TRUE, nulls = TRUE)` :

Return the entry corresponding to this ID. You can pass a vector of IDs, and you will get a list of entries.

id: A character vector containing entry identifiers.

drop: If set to TRUE and only one entry is requrested, then the returned value will be a single BiodbEntry object, otherwise it will be a list of BiodbEntry objects.

nulls: If set to TRUE, NULL entries are preserved. This ensures that the output list has the same length than the input vector 'id'. Otherwise they are removed from the final list.

Returned value: A list of BiodbEntry objects, the same size of the vector of IDs. The list will contain NULL values for invalid IDs. If drop is set to TRUE and only one etrny was requested then a single BiodbEntry is returned instead of a list.

`getEntryContent(id)` :

Get the contents of database entries from IDs (accession numbers).

id: A character vector of entry IDs.

Returned values: A character vector containing the contents of the requested IDs. If no content is available for an entry ID, then NA will be used.

`getEntryContentFromDb(entry.id)` :

Get the contents of entries directly from the database. A direct request or an access to the database will be made in order to retrieve the contents. No access to the biodb cache system will be made.

entry.id: A character vector with the IDs of entries to retrieve.

Returned value: A character vector, the same size of entry.id, with contents of the requested entries. An NA value will be set for the content of each entry for which the retrieval failed.

`getEntryIds(max.results = 0, ...)` :

Get entry identifiers from the database. More arguments can be given, depending on implementation in specific databases. For mass databases, the ones derived from BiodbBiodbMassdbConn class, the ms.level argument can be set.

max.results: The maximum of elements to return from the method.

...: First arguments to be passed to private .doGetEntryIds() method.

Returned value: A character vector containing entry IDs from the database. An empty vector for a remote database may mean that the database does not support requesting for entry accessions.

`getId()` :

Get the identifier of this connector.

Returned value: The identifier of this connector.

`getNbEntries(count = FALSE)` :

Get the number of entries contained in this database.

count: If set to TRUE and no straightforward way exists to get number of entries, count the output of getEntryIds().

Returned value: The number of entries in the database, as an integer.

`getSearchableFields()` :

Get the list of all searchable fields.

Returned value: A character vector containing all searchable fields for this connector.

`isCompounddb()` :

Tests if the connector's database is a compound database (i.e.: the connector class inherits from BiodbCompounddbConn class).

Returned value: Returns TRUE if the database is a compound database.

`isDownloadable()` :

Tests if the connector can download the database (i.e.: the connector class implements the interface BiodbDownloadable).

Returned value: Returns TRUE if the database is downloadable.

`isEditable()` :

Tests if this connector is able to edit the database (i.e.: the connector class implements the interface BiodbEditable). If this connector is editable, then you can call allowEditing() to enable editing.

Returned value: Returns TRUE if the database is editable.

`isMassdb()` :

Tests if the connector's database is a mass spectra database (i.e.: the connector class inherits from BiodbMassdbConn class).

Returned value: Returns TRUE if the database is a mass database.

`isRemotedb()` :

Tests of the connector is connected to a remote database (i.e.: the connector class inherits from BiodbRemotedbConn class).

Returned value: Returns TRUE if the database is a remote database.

`isSearchableByField(field)` :

Tests if a field can be used to search entries when using methods searchByName() and searchCompound().

field: The name of the field.

Returned value: Returns TRUE if the database is searchable using the specified field, FALSE otherwise.

`isWritable()` :

Tests if this connector is able to write into the database (i.e.: the connector class implements the interface BiodbWritable). If this connector is writable, then you can call allowWriting() to enable writing.

Returned value: Returns TRUE if the database is writable.

`makeRequest(...)` :

Makes a BiodbRequest instance using the passed parameters, and set ifself as the associated connector.

...: Those parameters are passed to the initializer of BiodbRequest.

Returned value: The BiodbRequest instance.

`makesRefToEntry(id, db, oid, any = FALSE, recurse = FALSE)` :

Tests if some entry of this database makes reference to another entry of another database.

id: A character vector of entry IDs from the connector's database.

db: Another database connector.

oid: A entry ID from database db.

any: If set to TRUE, returns a single logical value: TRUE if any entry contains a reference to oid, FALSE otherwise.

recurse: If set to TRUE, the algorithm will follow all references to entries from other databases, to see if it can establish an indirect link to 'oid'.

Returned value: A logical vector, the same size as 'id', with TRUE for each entry making reference to 'oid', and FALSE otherwise.

`searchByName(name, max.results = 0)` :

This method is deprecated.

Use searchForEntries() instead.

`searchForEntries(fields = NULL, max.results = 0)` :

Searches the database for entries whose name matches the specified name. Returns a character vector of entry IDs.

fields: A list of fields on which to filter entries. To get a match, all fields must be matched (i.e.: logical AND). The keys of the list are the entry field names on which to filter, and the values are the filtering parameters. For character fields, the filter parameter is a character vector in which all strings must be found inside the field's value. For numeric fields, the filter parameter is either a list specifying a min-max range ('list(min=1.0, max=2.5)') or a value with a tolerance in delta ('list(value=2.0, delta=0.1)') or ppm ('list(value=2.0, ppm=1.0)').

max.results: If set, the number of returned IDs is limited to this number.

Returned value: A character vector of entry IDs whose name matches the requested name.

`show()` :

Prints a description of this connector.

Returned value: None.

See Also

Super class BiodbConnBase, BiodbFactory, BiodbRemotedbConn and BiodbMassdbConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a compound CSV file database
chebi.tsv <- system.file("extdata", "chebi_extract.tsv", package='biodb')

# Create a connector
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi.tsv)

# Get 10 identifiers from the database:
ids <- conn$getEntryIds(10)

# Get number of entries contained in the database:
n <- conn$getNbEntries()

# Terminate instance.
mybiodb$terminate()
```

`BiodbConnBase-class` *Base class of* `BiodbConn` *for encapsulating all needed information for database access.*

Description

This is the base class for `BiodbConn` and `BiodbDbInfo`. When defining a new connector class, your class must not inherit from `BiodbBaseConn` but at least from `BiodbConn` (or `BiodbRemoteConn` or any subclass of `BiodbConn`). Its main purpose is to store property values. Those values are initialized from YAML files. The default definition file is located inside the package in "inst/definitions.yml" and is loaded at Biodb startup. However you can define your own files and load them using the `BiodbMain::loadDefinitions()` method.

Details

Arguments to the contructor are:

other: Another object inheriting from `BiodbBaseConn`, and from which property values will be copied.

db.class: The class of the database (`"mass.csv.file"`, `"comp.csv.file"`, ...).

properties: Some properties to set at initialization.

Methods

`getBaseUrl()` Returns the base URL.

`getConnClass()` :

Gets the associated connector OOP class.

Returned value: Returns the connector OOP class.

`getConnClassName()` :

Gets the name of the associated connector OOP class.

Returned value: Returns the connector OOP class name.

`getDbClass()` :

Gets the Biodb name of the database associated with this connector.

Returned value: A character value containing the Biodb database name.

`getEntryClass()` :

Gets the associated entry class.

Returned value: Returns the associated entry class.

`getEntryClassName()` :

Gets the name of the associated entry class.

Returned value: Returns the name of the associated entry class.

`getEntryContentType()` Returns the entry content type.

`getEntryFileExt()` :

Returns the entry file extension used by this connector.

Returned value: A character value containing the file extension.

`getEntryIdField()` :
Gets the name of the corresponding database ID field in entries.
Returned value: Returns the name of the database ID field.

`getName()` Returns the full database name.

`getPropertyValue(name)` :
Gets a property value.
name: The name of the property.
Returned value: The value of the property.

`getPropSlots(name)` :
Gets the slot fields of a property.
name: The name of a property.
Returned value: Returns a character vector containing all slot names defined.

`getPropValSlot(name, slot)` :
Retrieve the value of a slot of a property.
name: The name of a property.
slot: The slot name inside the property.
Returned value: The value of the slot ‘slot‘ of the property ‘name‘.

`getSchedulerNParam()` Returns the N parameter for the scheduler.

`getSchedulerTParam()` Returns the T parameter for the scheduler.

`getToken()` Returns the access token.

`getUrl(name)` Returns a URL.

`getUrls()` Returns the URLs.

`getWsUrl()` Returns the web sevices URL.

`getXmlNs()` Returns the XML namespace.

`hasProp(name)` :
Tests if this connector has a property.
name: The name of the property to check.
Returned value: Returns true if the property ‘name‘ exists.

`hasPropSlot(name, slot)` :
Tests if a slot property has a specific slot.
name: The name of a property.
slot: The slot name to check.
Returned value: Returns TRUE if the property ‘name‘ exists and has the slot ‘slot‘ defined, and FALSE otherwise.

`isSlotProp(name)` :
Tests if a property is a slot property.
name: The name of a property.
Returned value: Returns TRUE if the property is a slot propert, FALSE otherwise.

`propExists(name)` :
Checks if property exists.
name: The name of a property.
Returned value: Returns TRUE if the property ‘name‘ exists, and FALSE otherwise.

`setBaseUrl(url)` Sets the base URL.

`setPropertyValue(name, value)` :
Sets the value of a property.
name: The name of the property.
value: The new value to set the property to.
Returned value: None.

`setPropValSlot(name, slot, value)` :
Set the value of the slot of a property.
name: The name of the property.
slot: The name of the property's slot.
value: The new value to set the property's slot to.
Returned value: None.

`setSchedulerNParam(n)` Sets the N parameter for the scheduler.

`setSchedulerTParam(t)` Sets the T parameter for the scheduler.

`setToken(token)` Sets the access token.

`setUrl(name, url)` Returns a URL.

`setWsUrl(ws.url)` Sets the web sevices URL.

`updatePropertiesDefinition(def)` :
Update the definition of properties.
def: A named list of property definitions. The names of the list must be the property names.
Returned value: None.

See Also

Sub-classes `BiodbDbInfo` and `BiodbConn`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Accessing BiodbConnBase methods when using a BiodbDbInfo object
dbinf <- mybiodb$getDbsInfo()$get('comp.csv.file')

# Test if a property exists
dbinf$hasProp('name')

# Get a property value
dbinf$getPropertyValue('name')

# Get a property value slot
dbinf$getPropValSlot('urls', 'base.url')

# Terminate instance.
mybiodb$terminate()
```

BiodbConnObserver-class *The observer class of connectors.*

Description

Classes inheriting this class are able to register themselves to a connector instance and receive messages from it. It is used by the scheduler class to receive updates about URLs and query frequencies.

See Also

BiodbConn.

BiodbCsvEntry-class *Entry class for content in CSV format.*

Description

This is an abstract class for handling database entries whose content is in CSV format.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbCsvEntry",
  methods=list(

    initialize=function() {
      super(sep="\t", na.strings=c("", "-"))
    }
))
```

BiodbDbInfo-class *A class for describing the characteristics of a database.*

Description

This class is used by BiodbDbsInfo for storing database characteristics, and returning them through the get() method. This class inherits from BiodbConnBase.

See Also

Parent class BiodbDbsInfo and super class BiodbConnBase.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a BiodbDbInfo object for a database:
mybiodb$getDbsInfo()$get('comp.csv.file')

# Terminate instance.
mybiodb$terminate()
```

BiodbDbsInfo-class *A class for describing the available databases.*

Description

The unique instance of this class is handle by the BiodbMain class and accessed through the getDbsInfo() method.

Methods

checkIsDefined(db.id) :

Checks if a database is defined. Throws an error if the specified id does not correspond to a defined database.

db.id: A character vector of database IDs.

Returned value: None.

define(def, package = "biodb") :

Define databases from a structured object, normally loaded from a YAML file.

def: A named list of database definitions. The names of the list will be the IDs of the databases.

package: The package to which belong the new definitions.

Returned value: None.

`get(db.id = NULL, drop = TRUE)` :

Gets information on a database.

db.id: Database IDs, as a character vector. If set to NULL, informations on all databases will be returned.

drop: If TRUE and only one database ID has been submitted, returns a single BiodbDbInfo instance instead of a list.

Returned value: A list of BiodbDbInfo instances corresponding to the specified database IDs.

`getAll()` :

Gets informations on all databases.

Returned value: A list of all BiodbDbInfo instances.

`getIds()` :

Gets the database IDs.

Returned value: A character vector containing all the IDs of the defined databases.

`isDefined(db.id)` :

Tests if a database is defined.

db.id: A database ID, as a character string.

Returned value: TRUE if the specified id corresponds to a defined database, FALSE otherwise.

See Also

BiodbMain and child class BiodbDbInfo.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Getting the entry content type of a database:
db.inf <- mybiodb$getDbsInfo()$get('comp.csv.file')
cont.type <- db.inf$getPropertyValue('entry.content.type')

# Terminate instance.
mybiodb$terminate()
```

BiodbDownloadable-class

An abstract class (more like an interface) to model a remote database that can be downloaded locally.

Description

The class must be inherited by any remote database class that allows download of its whole content. The hidden/private method `.doDownload()` must be implemented by the database class.

Methods

`download()` :

Downloads the database content locally.

Returned value: None.

`getDownloadPath()` :

Gets the path where the downloaded content is written.

Returned value: The path where the downloaded database is written.

`isDownloaded()` :

Tests if the database has been downloaded.

Returned value: TRUE if the database content has already been downloaded.

`isExtracted()` :

Tests if the downloaded database has been extracted (in case the database needs extraction).

Returned value: TRUE if the downloaded database content has been extracted, FALSE otherwise.

See Also

BiodbRemotedbConn.

Examples

```
# Creating a connector class for a downloadable database:
FooDownloadableDb <- methods::setRefClass('FooDownloadableDb',
    contains=c('BiodbRemotedbConn', 'BiodbDownloadable'),

methods=list(

   # ... other methods ...

   # BiodbDownloadable methods
   .doDownload=function() {
       # Download the database
   },

   .doExtractDownload=function() {
       # Extract data from the downloaded file(s)
   }
))
```

`BiodbEditable-class` *An interface to model an editable database.*

Description

A database class that implements this interface allows the addition of new entries, the removal of entries and the modification of entries. If you want your modifications to be persistent, the database class must also be writable (see interface `BiodbWritable`), you must call the method `write` on the connector.

Methods

`addNewEntry(entry)` :

Adds a new entry to the database. The passed entry must have been previously created from scratch using BiodbFactory::createNewEntry() or cloned from an existing entry using BiodbEntry::clone().

entry: The new entry to add. It must be a valid BiodbEntry object.

Returned value: None.

`allowEditing()` :

Allows editing for this database.

Returned value: None.

`disallowEditing()` :

Disallows editing for this database.

Returned value: None.

`editingIsAllowed()` :

Tests if editing is allowed.

Returned value: TRUE if editing is allowed for this database, FALSE otherwise.

`setEditingAllowed(allow)` :

Allow or disallow editing for this database.

allow: A logical value.

Returned value: None.

See Also

`BiodbConn` and `BiodbWritable`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Create an empty MASS SQLite database
mydb <- mybiodb$getFactory()$createConn('mass.sqlite')

# Create new entry object
entry <- mybiodb$getFactory()$createNewEntry('mass.sqlite')
entry$setFieldValue('accession', '0')
entry$setFieldValue('name', 'Some Entry')

# Add the new entry
mydb$allowEditing()
```

```
mydb$addNewEntry(entry)

# Terminate instance.
mybiodb$terminate()
mybiodb <- NULL
```

`BiodbEntry-class` *The mother abstract class of all database entry classes.*

Description

An entry is an element of a database, identifiable by its accession number. Each contains a list of fields defined by a name and a value. The details of all fields that can be set into an entry are defined inside the class `BiodbEntryFields`. From this class are derived other abstract classes for different types of entry contents: `BiodbTxtEntry`, `BiodbXmlEntry`, `BiodbCsvEntry`, `BiodbJsonEntry` and `BiodbHtmlEntry`. Then concrete classes are derived for each database: `CompCsvEntry`, `MassCsvEntry`, etc. For biodb users, there is no need to know this hierarchy; the knowledge of this class and its methods is sufficient.

Methods

`appendFieldValue(field, value)` :

Appends a value to an existing field. If the field is not defined for this entry, then the field will be created and set to this value. Only fields with a cardinality greater than one can accept multiple values.

field: The name of a field.

value: The value to append.

Returned value: None.

`clone(db.class = NULL)` :

Clones this entry.

db.class: The database class (the Biodb database ID) of the clone. By setting this parameter, you can specify a different database for the clone, so you may clone an entry into another database if you wish. By default the class of the clone will be the same as the original entry.

Returned value: The clone, as a new BiodbEntry instance.

`computeFields(fields = NULL)` :

Computes fields. Look at all missing fields, and try to compute them using references to other databases, if a rule exists.

fields: A list of fields to review for computing. By default all fields will be reviewed.

Returned value: TRUE if at least one field was computed successfully, FALSE otherwise.

`getDbClass()` :

Gets the ID of the database associated with this entry.

Returned value: The name of the database class associated with this entry.

`getFieldDef(field)` :

Gets the definition of an entry field.

field: The name of the field.

return: A object BiodbEntryField which defines the field.

`getFieldNames()` :

Gets a list of all fields defined for this entry.

Returned value: A character vector containing all field names defined in this entry.

`getFieldsAsDataframe( only.atomic = TRUE, compute = TRUE, fields = NULL, fields.type = NULL, flatten = TRUE`
:

Converts this entry into a data frame.

only.atomic: If set to TRUE, only export field's values that are atomic (i.e.: of type vector).

compute: If set to TRUE and a field is not defined, try to compute it using internal defined computing rules. If set to FALSE, let the field undefined.

fields: Set to character vector of field names in order to restrict execution to this set of fields.

fields.type: If set, output all the fields of the specified type.

flatten: If set to TRUE and a field's value is a vector of more than one element, then export the field's value as a single string composed of the field's value concatenated and separated by the character defined in the 'multival.field.sep' config key. If set to FALSE or the field contains only one value, changes nothing.

limit: The maximum number of field values to write into new columns. Used for fields that can contain more than one value.

only.card.one: If set to TRUE, only fields with a cardinality of one will be extracted.

own.id: If set to TRUE includes the database id field named '<database_name>.id' whose values are the same as the 'accession' field.

duplicate.rows: If set to TRUE and merging field values with cardinality greater than one, values will be duplicated.

sort: If set to TRUE sort the order of columns alphabetically, otherwise do not sort.

virtualFields: If set to TRUE includes also virtual fields, otherwise excludes them.

Returned value: A data frame containg the values of the fields.

`getFieldsAsJson(compute = TRUE)` :

Converts this entry into a JSON string.

compute: If set to TRUE and a field is not defined, try to compute it using internal defined computing rules. If set to FALSE, let the field undefined.

Returned value: A JSON object from jsonlite package.

`getFieldsByType(type)` :

Gets the fields of this entry that have the specified type.

Returned value: A character vector containing the field names.

`getFieldValue( field, compute = TRUE, flatten = FALSE, last = FALSE, limit = 0, withNa = TRUE, duplicatedVal`
:

Gets the value of the specified field.

field: The name of a field.

compute: If set to TRUE and a field is not defined, try to compute it using internal defined computing rules. If set to FALSE, let the field undefined.

flatten: If set to TRUE and a field's value is a vector of more than one element, then export the field's value as a single string composed of the field's value concatenated and separated by the character defined in the 'multival.field.sep' config key. If set to FALSE or the field contains only one value, changes nothing.

last: If set to TRUE and a field's value is a vector of more than one element, then export only the last value. If set to FALSE, changes nothing.

limit: The maximum number of values to get in case the field contains more than one value.

withNa: If set to TRUE, keep NA values. Otherwise filter out NAs values in vectors.

Returned value: The value of the field.

`getId()` :

Gets the entry ID.

Returned value: the entry ID, which is the value if the 'accession' field.

`getName()` :

Gets a short text describing this entry instance.

Returned value: A character value concatenating the connector name with the entry accession.

`hasField(field)` :

Tests if a field is defined in this entry.

field: The name of a field.

Returned value: TRUE if the specified field is defined in this entry, FALSE otherwise.

`isNew()` :

Tests if this entry is new.

Returned value: TRUE if this entry was newly created, FALSE otherwise.

`makesRefToEntry(db, oid, recurse = FALSE)` :

Tests if this entry makes reference to another entry.

db: Another database connector.

oid: A entry ID from database db.

recurse: If set to TRUE, the algorithm will follow all references to entries from other databases, to see if it can establish an indirect link to 'oid'.

Returned value: TRUE if this entry makes reference to the entry oid from database db, FALSE otherwise.

`parentIsAConnector()` :

Tests if the parent of this entry is a connector instance.

Returned value: TRUE if this entry belongs to a connector, FALSE otherwise.

`parseContent(content)` :

Parses content string and set values accordingly for this entry's fields. This method is called automatically and should be run directly by users.

content: A character string containing definition for an entry and obtained from a database. The format can be: CSV, HTML, JSON, XML, or just text.

Returned value: None.

`removeField(field)` :

Removes the specified field from this entry.

field: The name of a field.

Returned value: None.

`setFieldValue(field, value)` :

Sets the value of a field. If the field is not already set for this entry, then the field will be created. See BiodbEntryFields for a list of possible fields in biodb.

field: The name of a field.

value: The value to set.

Returned value: None.

See Also

BiodbFactory, BiodbConn, BiodbEntryFields.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a compound CSV file database
chebi.tsv <- system.file("extdata", "chebi_extract.tsv", package='biodb')

# Get the connector of a compound database
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi.tsv)

# Get an entry:
entry <- conn$getEntry(conn$getEntryIds(1))

# Get all defined fields:
entry$getFieldNames()

# Get a field value:
accession <- entry$getFieldValue('accession')

# Test if a field is defined:
if (entry$hasField('name'))
  print(paste("The entry's name is ", entry$getFieldValue('name'),
  '.', sep=''))

# Export an entry as a data frame:
df <- entry$getFieldsAsDataframe()

# You can set or reset a field's value:
entry$setFieldValue('mass', 1893.1883)

# Terminate instance.
mybiodb$terminate()
```

`BiodbEntryField-class` *A class for describing an entry field.*

Description

This class is used by `BiodbEntryFields` for storing field characteristics, and returning them through the `get()` method. The constructor is not meant to be used, but for development purposes the constructor's parameters are nevertheless described in the Fields section.

Details

The constructor accepts the following arguments:

name: The name of the field.

alias: A character vector containing zero or more aliases for the field.

type: A type describing the field. One of: "mass", "name" or "id". Optional.

class: The class of the field. One of: "character", "integer", "double", "logical", "object", "data.frame".

card: The cardinality of the field: either "1" or "*".

forbids.duplicates: If set to TRUE, the field forbids duplicated values.

description: A description of the field.

allowed.values: The values authorized for the field.

lower.case: Set to TRUE if you want all values set to the field to be forced to lower case.

case.insensitive: Set to TRUE of you want the field to ignore case when checking a value.

computable.from: The Biodb ID of a database, from which this field can be computed.

virtual: If set to `TRUE`, the field is computed from other fields, and thus cannot be modified.

virtual.group.by.type: For a virtual field of class data.frame, this indicates to gather all fields of the specified type to build a data frame.

Methods

`addAlias(alias)` :

Adds an alias to the list of aliases.

alias: The name of a valid alias.

Returned value: None.

`addAllowedValue(key, value)` :

Adds an allowed value, as a synonym to already an existing value. Note that not all enumerate fields accept synonyms.

key: The key associated with the value (i.e.: the key is the main name of an allowed value).

value: The new value to add.

Returned value: None.

`addComputableFrom(directive)` :

Adds a directive from the list of computableFrom.

directive: A valid "computable from" directive.

Returned value: None.

`checkValue(value)` :

Checks if a value is correct. Fails if ‘value‘ is incorrect.

value: The value to check.

Returned value: None.

`correctValue(value)` :

Corrects a value so it is compatible with this field.

value: A value.

Returned value: The corrected value.

`equals(other, fail = FALSE)` :

Compares this instance with another, and tests if they are equal.

other: Another BiodbEntryField instance.

fail: If set to TRUE, then throws error instead of returning FALSE.

Returned value: TRUE if they are equal, FALSE otherwise.

`forbidsDuplicates()` :

Tests if this field forbids duplicates.

Returned value: TRUE if this field forbids duplicated values, FALSE otherwise.

`getAliases()` :

Get aliases.

Returned value: The list of aliases if some are defined, otherwise returns NULL.

`getAllNames()` :

Gets all names.

Returned value: The list of all names (main name and aliases).

`getAllowedValues(value = NULL)` :

Gets allowed values.

value: If this parameter is set to particular allowed values, then the method returns a list of synonyms for this value (if any).

Returned value: A character vector containing all allowed values.

`getDataFrameGroup()` :

Gets the defined data frame group, if any.

Returned value: The data frame group, as a character value.

`getDescription()` :

Get field’s description.

Returned value: The description of this field.

`getName()` :

Gets the name.

Returned value: The name of this field.

`getType()` :
Gets field's type.
Returned value: The type of this field.

`getVirtualGroupByType()` :
Gets type for grouping field values when building a virtual data frame.
Returned value: The type, as a character value.

`hasAliases()` :
Tests if this field has aliases.
Returned value: TRUE if this entry field defines aliases, FALSE otherwise.

`hasCardMany()` :
Tests if this field has a cardinality greater than one.
Returned value: TRUE if the cardinality of this field is many, FALSE otherwise.

`hasCardOne()` :
Tests if this field has a cardinality of one.
Returned value: TRUE if the cardinality of this field is one, FALSE otherwise.

`isAtomic()` :
Tests if this field's type is an atomic type.
Returned value: TRUE if the field's type is vector (i.e.: character, integer, double or logical), FALSE otherwise.

`isCaseInsensitive()` :
Tests if this field is case sensitive.
Returned value: TRUE if this field is case insensitive, FALSE otherwise.

`isComputable()` :
Tests if this field is computable from another field or another database.
Returned value: TRUE if the field is computable, FALSE otherwise.

`isComputableFrom()` :
Gets the ID of the database from which this field can be computed.
Returned value: The list of databases where to find this field's value.

`isDataFrame()` :
Tests if this field's type is 'data.frame'.
Returned value: TRUE if field's type is data frame, FALSE otherwise.

`isEnumerate()` :
Tests if this field is an enumerate type (i.e.: it defines allowed values).
Returned value: TRUE if this field defines some allowed values, FALSE otherwise.

`isObject()` :
Tests if this field's type is a class.
Returned value: TRUE if field's type is a class, FALSE otherwise.

`isVector()` :
Tests if this field's type is a basic vector type.
Returned value: TRUE if the field's type is vector (i.e.: character, integer, double or logical), FALSE otherwise.

`isVirtual()` :

Tests if this field is a virtual field.

Returned value: TRUE if this field is virtual, FALSE otherwise.

`removeAlias(alias)` :

Removes an alias from the list of aliases.

alias: The name of a valid alias.

Returned value: None.

`removeComputableFrom(directive)` :

Removes a directive from the list of computableFrom.

directive: A valid "computable from" directive.

Returned value: None.

`updateWithValuesFrom(other)` :

Updates fields using values from ‘other‘ instance. The updated fields are: ’alias’ and ’computable.from’. No values will be removed from those vectors. The new values will only be appended. This allows to extend an existing field inside a new connector definition.

other: Another BiodbEntryField instance.

Returned value: None.

See Also

Parent class BiodbEntryFields.

Examples

```
# Get the class of the InChI field.
mybiodb <- biodb::newInst()
inchi.field.class <- mybiodb$getEntryFields()$get('inchi')$getClass()

# Test the cardinality of a field
card.one <- mybiodb$getEntryFields()$get('name')$hasCardOne()
card.many <- mybiodb$getEntryFields()$get('name')$hasCardMany()

# Get the description of a field
desc <- mybiodb$getEntryFields()$get('inchi')$getDescription()

# Terminate instance.
mybiodb$terminate()
```

BiodbEntryFields-class

A class for handling description of all entry fields.

Description

The unique instance of this class is handle by the BiodbMain class and accessed through the `getEntryFields()` method.

Methods

`checkIsDefined(name)` :

Tests if names are valid defined fields. Throws an error if any name does not correspond to a defined field.

name: A character vector of names or aliases to test.

Returned value: None.

`define(def)` :

Defines fields.

def: A named list of field definitions. The names of the list are the main names of the fields.

Returned value: None.

`formatName(name)` :

Format field name(s) for biodb format: set to lower case and remove dot or underscore characters depending on configuration.

name: A character vector of names or aliases to test.

Returned value: A character vector of formatted names.

`get(name, drop = TRUE)` :

Gets a BiodbEntryField instance.

name: A character vector of names or aliases.

drop: If TRUE and only one name has been submitted, returns a single BiodbEntryField instance instead of a list.

Returned value: A named list of BiodbEntryField instances. The names of the list are the real names of the entry fields, thus they may be different from the one provided inside the name argument.

`getDatabaseIdField(database)` :

Gets a database ID field.

database: The name (i.e.: Biodb ID) of a database.

Returned value: The name of the field handling identifiers (i.e.: accession numbers) for this database.

`getFieldNames(type = NULL, computable = NULL)` :

Gets the main names of all fields.

type: Set this parameter to a character vector in order to return only the names of the fields corresponding to the types specified.

computable: If set to TRUE, returns only the names of computable fields. If set to FALSE, returns only the names of fields that are not computable.

Returned value: A character vector containing all selected field names.

`getRealName(name, fail = TRUE)` :

Gets the real names (main names) of fields. If some name is not found neither in aliases nor in real names, an error is thrown.

name: A character vector of names or aliases.

fail: Fails if name is unknown.

Returned value: A character vector, the same length as ‘name‘, with the real field name for each name given (i.e.: each alias is replaced with the real name).

`isAlias(name)` :

Tests if names are aliases.

name: A character vector of names or aliases to test.

Returned value: A logical vector, the same length as 'name', with TRUE for name values that are an alias of a field, and FALSE otherwise.

`isDefined(name)` :

Tests if names are defined fields.

name: A character vector of names or aliases to test.

Returned value: A logical vector, the same length as 'name', with TRUE for name values that corresponds to a defined field.

See Also

`BiodbMain` and child class `BiodbEntryField`.

Examples

```
# Getting information about the accession field:
mybiodb <- biodb::newInst()
entry.field <- mybiodb$getEntryFields()$get('accession')

# Test if a name is an alias of a field
mybiodb$getEntryFields()$isAlias('genesymbols')

# Test if a name is associated with a defined field
mybiodb$getEntryFields()$isDefined('name')

# Terminate instance.
mybiodb$terminate()
```

BiodbFactory-class *A class for constructing biodb objects.*

Description

This class is responsible for the creation of database connectors and database entries. You must go through the single instance of this class to create and get connectors, as well as instantiate entries. To get the single instance of this class, call the `getFactory()` method of class `BiodbMain`.

Methods

`connExists(conn.id)` :

Tests if a connector exists.

conn.id: A connector ID.

Returned value: TRUE if a connector with this ID exists, FALSE otherwise.

`createConn( db.class, url = NULL, token = NA_character_, fail.if.exists = TRUE, get.existing.conn = TRUE,`
:

Creates a connector to a database.

db.class: The type of a database. The list of types can be obtained from the class BiodbDbsInfo.

url: An URL to the database for which to create a connection. Each database connector is configured with a default URL, but some allow you to change it.

token: A security access token for the database. Some database require such a token for all or some of their webservices. Usually you obtain the token through your account on the database website.

fail.if.exists: If set to TRUE, the method will fail if a connector for

get.existing.conn: This argument will be used only if fail.if.exists is set to FALSE and an identical connector already exists. If it set to TRUE, the existing connector instance will be returned, otherwise NULL will be returned.

conn.id: If set, this identifier will be used for the new connector. An error will be raised in case another connector already exists with this identifier.

cache.id: If set, this ID will be used as the cache ID for the new connector. An error will be raised in case another connector already exists with this cache identifier.

Returned value: An instance of the requested connector class.

`createNewEntry(db.class)` :

Creates a new entry from scratch. This entry is not stored in cache.

db.class: A database ID.

Returned value: A new BiodbEntry object.

`deleteAllCacheEntries(conn.id)` :

Deletes all entries stored in the cache of the given connector.

conn.id: A connector ID.

Returned values: None.

`deleteAllConnectors()` :

Deletes all connectors.

Returned value: None.

`deleteAllEntriesFromVolatileCache(conn.id)` :

Deletes all entries stored in the cache of the given connector. This method is deprecated, please use deleteAllEntriesFromVolatileCache() instead.

conn.id: A connector ID.

Returned values: None.

`deleteConn(conn)` :

Deletes an existing connector.

conn.id: A connector instance or a connector ID.

Returned value: None.

`deleteConnByClass(db.class)` :

Deletes all existing connectors from a same class.

db.class: The type of a database. All connectors of this database type will be deleted.

Returned value: None.

`getAllCacheEntries(conn.id)` :

For a connector, gets all entries stored in the cache.

conn.id: A connector ID.

Returned values: A list of BiodbEntry objects.

`getAllConnectors()` :

Gets all connectors.

Returned value: A list of all created connectors.

`getConn(conn.id, class = TRUE, create = TRUE)` :

Gets an instantiated connector instance, or create a new one.

conn.id: An existing connector ID.

class: If set to TRUE, and "conn.id" does not correspond to any instantiated connector, then interpret "conn.id" as a database class and looks for the first instantiated connector of that class.

create: If set to TRUE, and "class" is also set to TRUE, and no suitable instantiated connector was found, then creates a new connector instance of the class specified by "conn.id".

Returned value: The connector instance corresponding to the connector ID or to the database ID submitted (if class "parameter" is set to TRUE).

`getEntry(conn.id, id, drop = TRUE)` :

Retrieves database entry objects from IDs (accession numbers), for the specified connector.

conn.id: An existing connector ID.

id: A character vector containing database entry IDs (accession numbers).

drop: If set to TRUE and the list of entries contains only one element, then returns this element instead of the list. If set to FALSE, then returns always a list.

Returned value: A list of BiodbEntry objects, the same length as ‘id‘. A NULL value is put into the list for each invalid ID of ‘id‘.

See Also

BiodbMain, BiodbConn and BiodbEntry.

Examples

```
# Create a BiodbMain instance with default settings:
mybiodb <- biodb::newInst()

# Obtain the factory instance:
factory <- mybiodb$getFactory()

# Get a compound CSV file database
chebi.tsv <- system.file("extdata", "chebi_extract.tsv", package='biodb')

# Create a connector:
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi.tsv)

# Get a database entry:
entry <- conn$getEntry(conn$getEntryIds(1))
```

```
# Terminate instance.
mybiodb$terminate()
```

BiodbHtmlEntry-class *Entry class for content in HTML format.*

Description

This is an abstract class for handling database entries whose content is in HTML format.

See Also

Super class BiodbXmlEntry.

Examples

```
# Create a concrete entry class inheriting from this class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbHtmlEntry")
```

BiodbJsonEntry-class *Entry class for content in JSON format.*

Description

This is an abstract class for handling database entries whose content is in JSON format.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbJsonEntry")
```

BiodbListEntry-class *Entry class for content in list format.*

Description

This is an abstract class for handling database entries whose content is in list format.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbListEntry")
```

BiodbMain-class *The central class of the biodb package.*

Description

The main class of the biodb package. In order to use the biodb package, you need first to create an instance of this class.

Details

The constructor takes a single argument, autoloadExtraPkgs, to enable (TRUE or default) or disable (FALSE) autoloading of extra biodb packages.

Once the instance is created, some other important classes (BiodbFactory, BiodbPersistentCache, BiodbConfig, ...) are instantiated (just once) and their instances are later accessible through get*() methods.

Methods

addColsToDataframe(x, id.col, db, fields, limit = 3, prefix = "") :

Using

x: A data frame containing at least one column with Biodb entry IDs identified by the parameter 'id.col'.

id.col: The name of the column containing IDs inside the input data frame.

db: The biodb database name for the entry IDs, or a connector ID, as a sinle character value.

fields: A character vector containing entry fields to add.

limit: The maximum number of field values to write into new columns. Used for fields that can contain more than one value. Set it to 0 to get all values.

prefix: Insert a prefix at the start of all field names.

Returned value: A data frame containing 'x' and new columns appended for the fields requested.

`addObservers(observers)` :

Adds new observers. Observers will be called each time an event occurs. This is the way used in biodb to get feedback about what is going inside biodb code.

observers: Either a `BiodbObserver` instance or a list of `BiodbObserver` instances.

Returned value: None.

`collapseRows(x, sep = "|", cols = 1L)` :

Collapses rows of a data frame, by looking for duplicated values in the reference columns (parameter 'cols'). The values contained in the reference columns are supposed to be ordered inside the data frame, in the sens that all duplicated values are supposed to directly follow the original values. For all rows containing duplicated values, we look at values in all other columns and concatenate values in each column containing different values.

x: A data frame.

cols: The indices or the names of the columns used as reference.

sep: The separator to use when concatenating values in collapsed rows.

Returned value: A data frame, with rows collapsed.

`computeFields(entries)` :

Computes missing fields in entries, for those fields that are comptable.

entries: A list of `BiodbEntry` instances.

Returned value: None.

`convertEntryIdFieldToDbClass(entry.id.field)` :

Gets the database class name corresponding to an entry ID field.

entry.id.field: The name of an ID field. It must end with ".id".

`copyDb(conn.from, conn.to, limit = 0)` :

Copies all entries of a database into another database. The connector of the destination database must be editable.

conn.from: The connector of the source datababase to copy.

conn.to: The connector of the destination database.

limit: The number of entries of the source database to copy. If set to `NULL`, copy the whole database.

Returned value: None.

`entriesFieldToVctOrLst( entries, field, flatten = FALSE, compute = TRUE, limit = 0, withNa = TRUE )` :

Extracts the value of a field from a list of entries. Returns either a vector or a list depending on the type of the field.

entries: A list of `BiodbEntry` instances.

field: The name of a field.

flatten: If set to `TRUE` and the field has a cardinality greater than one, then values be converted into a vector of class character in which each entry values are collapsed.

compute: If set to `TRUE`, computable fields will be output.

limit: The maximum number of values to retrieve for each entry. Set to 0 to get all values.

withNa: If set to TRUE, keep NA values. Otherwise filter out NAs values in vectors.

Returned value: A vector if the field is atomic or flatten is set to TRUE, otherwise a list.

`entriesToDataframe( entries, only.atomic = TRUE, null.to.na = TRUE, compute = TRUE, fields = NULL, limit =`
:

Converts a list of entries or a list of list of entries (BiodbEntry objects) into a data frame.

entries: A list of BiodbEntry instances or a list of list of BiodbEntry instances.

only.atomic: If set to TRUE, output only atomic fields, i.e.: the fields whose value type is one of integer, numeric, logical or character.

null.to.na: If set to TRUE, each NULL entry in the list is converted into a row of NA values.

compute: If set to TRUE, computable fields will be output.

fields: A character vector of field names to output. The data frame output will be restricted to this list of fields.

limit: The maximum number of field values to write into new columns. Used for fields that can contain more than one value. Set it to 0 to get all values.

drop: If set to TRUE and the resulting data frame has only one column, a vector will be output instead of data frame.

sort.cols: Sort columns in alphabetical order.

flatten: If set to TRUE, then each field with a cardinality greater than one, will be converted into a vector of class character whose values are collapsed.

only.card.one: Output only fields whose cardinality is one.

own.id: If set to TRUE includes the database id field named '<database_name>.id' whose values are the same as the 'accession' field.

prefix: Insert a prefix at the start of all field names.

Returned value: A data frame containing the entries. Columns are named according to field names.

`entriesToJson(entries, compute = TRUE)` :

Converts a list of BiodbEntry objects into JSON. Returns a vector of characters.

entries: A list of BiodbEntry instances.

compute: If set to TRUE, computable fields will added to JSON too.

Returned value: A list of JSON strings, the same length as entries list.

`entryIdsToDataframe( ids, db, fields = NULL, limit = 3, prefix = "", own.id = FALSE )` :

Construct a data frame using entry IDs and field values of the corresponding entries.

ids: A character vector of entry IDs or a list of character vectors of entry IDs.

db: The biodb database name for the entry IDs, or a connector ID, as a sinle character value.

fields: A character vector containing entry fields to add.

limit: The maximum number of field values to write into new columns. Used for fields that can contain more than one value. Set it to 0 to get all values.

own.id: If set to TRUE includes the database id field named '<database_name>.id' whose values are the same as the 'accession' field.

prefix: Insert a prefix at the start of all field names.

Returned value: A data frame containing in columns the requested field values, with one entry per line, in the same order than in 'ids' vector.

`fieldIsAtomic(field)` :

DEPRECATED method to test if a field is an atomic field. The new method is `BiodbEntryField::isVector()`.

`getConfig()` :

Returns the single instance of the `BiodbConfig` class.

Returned value: The instance of the `BiodbConfig` class attached to this BiodbMain instance.

`getDbsInfo()` :

Returns the single instance of the `BiodbDbsInfo` class.

Returned value: The instance of the `BiodbDbsInfo` class attached to this BiodbMain instance.

`getEntryFields()` :

Returns the single instance of the `BiodbEntryFields` class.

Returned value: The instance of the `BiodbEntryFields` class attached to this BiodbMain instance.

`getFactory()` :

Returns the single instance of the `BiodbFactory` class.

Returned value: The instance of the `BiodbFactory` class attached to this BiodbMain instance.

`getFieldClass(field)` :

DEPRECATED method to get the class of a field. The new method is `BiodbMain::getEntryFields()$get(field)$g`

`getObservers()` :

Gets the list of registered observers.

Returned value: The list or registered observers.

`getPersistentCache()` :

Returns the single instance of the BiodbPersistentCache class.

Returned value: The instance of the BiodbPersistentCache class attached to this BiodbMain instance.

`getRequestScheduler()` :

Returns the single instance of the `BiodbRequestScheduler` class.

Returned value: The instance of the `BiodbRequestScheduler` class attached to this BiodbMain instance.

`loadDefinitions(file, package = "biodb")` :

Loads databases and entry fields definitions from YAML file.

file: The path to a YAML file containing definitions for `BiodbMain` (databases, fields or configuration keys).

package: The package to which belong the new definitions.

Returned value: None.

`saveEntriesAsJson(entries, files, compute = TRUE)` :

Saves a list of entries in JSON format. Each entry will be saved in a separate file.

entries: A list of `BiodbEntry` instances.

files: A character vector of file paths, the same length as entries list.

compute: If set to `TRUE`, computable fields will be saved too.

Returned value: None.

`terminate()` :

Closes `BiodbMain` instance. Call this method when you are done with your `BiodbMain` instance.

Returned value: None.

See Also

BiodbFactory, BiodbPersistentCache, BiodbConfig, BiodbObserver, BiodbEntryFields, BiodbDbsInfo.

Examples

```
# Create an instance:
mybiodb <- biodb::newInst()

# Get the factory instance
fact <- mybiodb$getFactory()

# Terminate instance.
mybiodb$terminate()
mybiodb <- NULL
```

BiodbMassdbConn-class *The mother class of all Mass spectra databases.*

Description

All Mass spectra databases inherit from this class. It thus defines methods specific to mass spectrometry.

Methods

`collapseResultsDataFrame(results.df, mz.col = "mz", rt.col = "rt", sep = "|")` :

Collapse rows of a results data frame, by outputing a data frame with only one row for each MZ/RT value.

results.df: Results data frame.

mz.col: The name of the M/Z column in the results data frame.

rt.col: The name of the RT column in the results data frame.

sep: The separator used to concatenate values, when collapsing results data frame.

Returned value: A data frame with rows collapsed.

`filterEntriesOnRt( entry.ids, rt, rt.unit, rt.tol, rt.tol.exp, chrom.col.ids, match.rt )` :

Filters a list of entries on retention time values.

entry.ids: A character vector of entry IDs.

rt: A vector of retention times to match. Used if input.df is not set. Unit is specified by rt.unit parameter.

rt.unit: The unit for submitted retention times. Either 's' or 'min'.

rt.tol: The plain tolerance (in seconds) for retention times: input.rt - rt.tol <= database.rt <= input.rt + rt.tol.

rt.tol.exp: A special exponent tolerance for retention times: input.rt - input.rt ** rt.tol.exp <= database.rt <= input.rt + input.rt ** rt.tol.exp. This exponent is applied on the RT value in

seconds. If both rt.tol and rt.tol.exp are set, the inequality expression becomes: input.rt - rt.tol - input.rt ** rt.tol.exp <= database.rt <= input.rt + rt.tol + input.rt ** rt.tol.exp.

chrom.col.ids: IDs of chromatographic columns on which to match the retention time.

match.rt: If set to TRUE, filters on RT values, otherwise does not do any filtering.

Returned value: A character vector containing entry IDs after filtering.

`getChromCol(ids = NULL)` :

Gets a list of chromatographic columns contained in this database.

ids: A character vector of entry identifiers (i.e.: accession numbers). Used to restrict the set of entries on which to run the algorithm.

Returned value : A data.frame with two columns, one for the ID 'id' and another one for the title 'title'.

`getMatchingMzField()` :

Gets the field to use for M/Z matching.

Returned value: The name of the field (one of peak.mztheo or peak.mzexp).

`getMzValues(ms.mode = NULL, max.results = 0, precursor = FALSE, ms.level = 0)` :

Gets a list of M/Z values contained inside the database.

ms.mode: The MS mode. Set it to either 'neg' or 'pos' to limit the output to one mode.

max.results: If set, it is used to limit the size of the output.

precursor: If set to TRUE, then restrict the search to precursor peaks.

ms.level: The MS level to which you want to restrict your search. 0 means that you want to search in all levels.

Returned value: A numeric vector containing M/Z values.

`getNbPeaks(mode = NULL, ids = NULL)` :

Gets the number of peaks contained in the database.

mode: The MS mode. Set it to either 'neg' or 'pos' to limit the counting to one mode.

ids: A character vector of entry identifiers (i.e.: accession numbers). Used to restrict the set of entries on which to run the algorithm.

Returned value: The number of peaks, as an integer.

`msmsSearch( spectrum, precursor.mz, mz.tol, mz.tol.unit = c("plain", "ppm"), ms.mode, npmin = 2, dist.fun` :

Searches MSMS spectra matching a template spectrum. The mz.tol parameter is applied on the precursor search.

spectrum: A template spectrum to match inside the database.

precursor.mz: The M/Z value of the precursor peak of the mass spectrum.

mz.tol: The M/Z tolerance, whose unit is defined by mz.tol.unit.

mz.tol.unit: The type of the M/Z tolerance. Set it to either to 'ppm' or 'plain'.

ms.mode: The MS mode. Set it to either 'neg' or 'pos'.

npmin: The minimum number of peak to detect a match (2 is recommended).

dist.fun: The distance function used to compute the distance betweem two mass spectra.

msms.mz.tol: M/Z tolerance to apply while matching MSMS spectra. In PPM.

msms.mz.tol.min: Minimum of the M/Z tolerance (plain unit). If the M/Z tolerance computed with 'msms.mz.tol' is lower than 'msms.mz.tol.min', then 'msms.mz.tol.min' will be used.

max.results: If set, it is used to limit the number of matches found for each M/Z value.

Returned value: A data frame with columns 'id', 'score' and 'peak.*'. Each 'peak.*' column corresponds to a peak in the input spectrum, in the same order and gives the number of the peak that was matched with it inside the matched spectrum whose ID is inside the 'id' column.

`searchForMassSpectra( mz.min = NULL, mz.max = NULL, mz = NULL, mz.tol = NULL, mz.tol.unit = c("plain", "ppm`
:

Searches for entries (i.e.: spectra) that contain a peak around the given M/Z value. Entries can also be filtered on RT values. You can input either a list of M/Z values through mz argument and set a tolerance with mz.tol argument, or two lists of minimum and maximum M/Z values through mz.min and mz.max arguments.

mz: A vector of M/Z values.

mz.tol: The M/Z tolerance, whose unit is defined by mz.tol.unit.

mz.tol.unit: The type of the M/Z tolerance. Set it to either to 'ppm' or 'plain'.

mz.min: A vector of minimum M/Z values.

mz.max: A vector of maximum M/Z values. Its length must be the same as 'mz.min'.

rt: A vector of retention times to match. Used if input.df is not set. Unit is specified by rt.unit parameter.

rt.unit: The unit for submitted retention times. Either 's' or 'min'.

rt.tol: The plain tolerance (in seconds) for retention times: input.rt - rt.tol <= database.rt <= input.rt + rt.tol.

rt.tol.exp: A special exponent tolerance for retention times: input.rt - input.rt ** rt.tol.exp <= database.rt <= input.rt + input.rt ** rt.tol.exp. This exponent is applied on the RT value in seconds. If both rt.tol and rt.tol.exp are set, the inequality expression becomes: input.rt - rt.tol - input.rt ** rt.tol.exp <= database.rt <= input.rt + rt.tol + input.rt ** rt.tol.exp.

chrom.col.ids: IDs of chromatographic columns on which to match the retention time.

precursor: If set to TRUE, then restrict the search to precursor peaks.

min.rel.int: The minimum relative intensity, in percentage (i.e.: float number between 0 and 100).

ms.mode: The MS mode. Set it to either 'neg' or 'pos'.

ms.level: The MS level to which you want to restrict your search. 0 means that you want to search in all levels.

max.results: If set, it is used to limit the number of matches found for each M/Z value.

Returned value: A character vector of spectra IDs.

`searchMsEntries( mz.min = NULL, mz.max = NULL, mz = NULL, mz.tol = NULL, mz.tol.unit = c("plain", "ppm"), rt`
:

This method is deprecated.

Use searchForMassSpectra() instead.

`searchMsPeaks( input.df = NULL, mz = NULL, mz.tol = NULL, mz.tol.unit = c("plain", "ppm"), min.rel.int = 0,`
:

For each M/Z value, searches for matching MS spectra and returns the matching peaks.

input.df: A data frame taken as input for searchMsPeaks(). It must contain a columns 'mz', and optionaly an 'rt' column.

mz: A vector of M/Z values to match. Used if input.df is not set.

mz.tol: The M/Z tolerance, whose unit is defined by mz.tol.unit.

mz.tol.unit: The type of the M/Z tolerance. Set it to either to 'ppm' or 'plain'.

min.rel.int: The minimum relative intensity, in percentage (i.e.: float number between 0 and 100).

ms.mode: The MS mode. Set it to either 'neg' or 'pos'.

ms.level: The MS level to which you want to restrict your search. 0 means that you want to search in all levels.

max.results: If set, it is used to limit the number of matches found for each M/Z value.

chrom.col.ids: IDs of chromatographic columns on which to match the retention time.

rt: A vector of retention times to match. Used if input.df is not set. Unit is specified by rt.unit parameter.

rt.unit: The unit for submitted retention times. Either 's' or 'min'.

rt.tol: The plain tolerance (in seconds) for retention times: input.rt - rt.tol <= database.rt <= input.rt + rt.tol.

rt.tol.exp: A special exponent tolerance for retention times: input.rt - input.rt ** rt.tol.exp <= database.rt <= input.rt + input.rt ** rt.tol.exp. This exponent is applied on the RT value in seconds. If both rt.tol and rt.tol.exp are set, the inequality expression becomes: input.rt - rt.tol - input.rt ** rt.tol.exp <= database.rt <= input.rt + rt.tol + input.rt ** rt.tol.exp.

precursor: If set to TRUE, then restrict the search to precursor peaks.

precursor.rt.tol: The RT tolerance used when matching the precursor.

insert.input.values: Insert input values at the beginning of the result data frame.

prefix: Add prefix on column names of result data frame.

compute: If set to TRUE, use the computed values when converting found entries to data frame.

fields: A character vector of field names to output. The data frame output will be restricted to this list of fields.

fieldsLimit: The maximum of values to output for fields with multiple values. Set it to 0 to get all values.

input.df.colnames: Names of the columns in the input data frame.

match.rt: If set to TRUE, match also RT values.

Returned value: A data frame with at least input MZ and RT columns, and annotation columns prefixed with 'prefix' if set. For each matching found a row is output. Thus if n matchings are found for M/Z value x, then there will be n rows for x, each for a different match. The number of matching found for each M/Z value is limited to 'max.results'.

`searchMzRange( mz.min, mz.max, min.rel.int = 0, ms.mode = NULL, max.results = 0, precursor = FALSE, ms.leve`
: Find spectra in the given M/Z range. Returns a list of spectra IDs.

`searchMzTol( mz, mz.tol, mz.tol.unit = "plain", min.rel.int = 0, ms.mode = NULL, max.results = 0, precursor`
: Find spectra containg a peak around the given M/Z value. Returns a character vector of spectra IDs.

`setMatchingMzField(field = c("peak.mztheo", "peak.mzexp")) :`
: Sets the field to use for M/Z matching.

field: The field to use for matching.

Returned value: None.

See Also

Super class BiodbConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get connector
conn <- mybiodb$getFactory()$createConn('mass.csv.file')

# Terminate instance.
mybiodb$terminate()
```

`BiodbObserver-class` *The mother abstract class of all observer classes.*

Description

This abstract class defines all the methods that can be used to send messages to the observers. You can define new observer classes by inherting from this class.

Methods

`notifyProgress(what, index, total)` :

Notify about the progress of an action.

what: A short description of the action.

index: A positive integer number indicating the progress.

total: The maximum for "index". When reached, the action is completed.

Returned value: None.

`terminate()` :

Terminates the instance. This method will be called automatically by the BiodbMain instance when you call BiodbMain::terminate().

Returned value: None.

Examples

```
# Define a new observer class
MyObsClass <- methods::setRefClass("MyObsClass", contains='BiodbObserver',
  methods=list(notifyProgress=function(what, index, total) {
      sprintf("Progress for %s is %d / %d.", what, index, total)
  }
))

# Create an instance and register an instance of the new observer class:
mybiodb <- biodb::newInst()
mybiodb$addObservers(MyObsClass$new())

# Terminate instance.
mybiodb$terminate()
```

`BiodbPersistentCache-class` *A class for handling file caching.*

Description

This class manages a cache system for saving downloaded files and request results.

Details

It is designed for internal use, but you can still access some of the read-only methods if you wish.

Inside the cache folder, one folder is created for each cache ID (each remote database has one single cache ID, always the same ,except if you change its URL). In each of theses folders are stored the cache files for this database.

Methods

`deleteAllFiles(cache.id, fail = FALSE, prefix = FALSE)` :

Deletes, in the cache system, all files associated with this cache ID.

cache.id: The cache ID to use.

prefix: If set to TRUE, use cache.id as a prefix, deleting all files whose cache.id starts with this prefix.

fail: If set to TRUE, a warning will be emitted if no cache files exist for this cache ID.

Returned value: None.

`deleteFile(cache.id, name, ext)` :

Deletes a list of files inside the cache system.

cache.id: The cache ID to use.

name: A character vector containing file names.

ext: The extension of the files, without the dot ("html", "xml", etc).

Returned value: None.

`deleteFiles(cache.id, ext)` :

Deletes all files with the specific extension of the cache ID in the cache system.

cache.id: The cache ID to use.

ext: The extension of the files, without the dot ("html", "xml", etc). Only files having this extension will be deleted.

Returned value: None.

`disable()` :

DEPRECATED method. Use now `BiodbConfig::disable('cache.system')`.

`enable()` :

DEPRECATED method. Use now `BiodbConfig::enable('cache.system')`.

`enabled()` :

DEPRECATED method. Use now `BiodbConfig::isEnabled('cache.system')`.

`erase()` :

Erases the whole cache.

Returned value: None.

`fileExist(cache.id, name, ext)` :

Tests if a particular file exist in the cache.

cache.id: The cache ID to use.

name: A character vector containing file names.

ext: The extension of the files, without the dot ("html", "xml", etc).

Returned value: A logical vector, the same size as `name`, with `TRUE` value if the file exists in the cache, or `FALSE` otherwise.

`filesExist(cache.id)` :

Tests if at least one cache file exist for the specified cache ID.

cache.id: The cache ID to use.

Returned value: A single boolean value.

`getDir()` :

Gets the absolute path to the cache directory.

Returned value: The absolute path of the cache directory.

`getFilePath(cache.id, name, ext)` :

Gets path of file in cache system.

cache.id: The cache ID to use.

name: A character vector containing file names.

ext: The extension of the files.

Returned value: A character vector, the same size as `names`, containing the paths to the files.

`getFolderPath(cache.id)` :

Gets path to the cache system sub-folder dedicated to this cache ID.

cache.id: The cache ID to use.

Returned value: A string containing the path to the folder.

`getTmpFolderPath()` :

Gets path to the cache system temporary folder.

Returned value: A string containing the path to the folder.

`getUsedCacheIds()` :

Returns a list of cache IDs actually used to store cache files.

Returned value: A character vector containing all the cache IDs actually used inside the cache system.

`getVersion()` :

Returns the cache version.

Returned value: The current cache version.

`isReadable(conn = NULL)` :

Checks if the cache system is readable.

conn: If not `NULL`, checks if the cache system is readable for this particular connector.

Returned value: `TRUE` if the cache system is readable, `FALSE` otherwise.

`isWritable(conn = NULL)` :

Checks if the cache system is writable.

conn: If not `NULL`, checks if the cache system is writable for this particular connector.

Returned value: `TRUE` if the cache system is writable, `FALSE` otherwise.

`listFiles( cache.id, ext = NA_character_, extract.name = FALSE, full.path = FALSE )` :

Lists files present in the cache system.

cache.id: The cache ID to use.

ext: The extension of the files, without the dot ("html", "xml", etc).

extract.name: If set to `TRUE`, instead of returning the file paths, returns the list of names used to construct the file name: [cache_folder]/[cache.id]/[name].[ext].

full.path: If set to `TRUE`, returns full path for files.

Returned value: The files of found files, or the names of the files if `extract.name` is set to `TRUE`.

`loadFileContent(cache.id, name, ext, output.vector = FALSE)` :

Loads content of files from the cache.

cache.id: The cache ID to use.

name: A character vector containing file names.

ext: The extension of the files.

output.vector: If set to `TRUE`, force output to be a `vector` instead of a `list`. Where the list contains a `NULL`, the `vector` will contain an `NA` value.

Returned value: A list (or a vector if `output.vector` is set to `TRUE`), the same size as `name`, containing the contents of the files. If some file does not exist, a `NULL` value is inserted inside the list.

`markerExist(cache.id, name)` :

Tests if markers exist in the cache. Markers are used, for instance, by biodb to remember that a downloaded zip file from a database has been extracted correctly.

cache.id: The cache ID to use.

name: A character vector containing marker names.

Returned value: A logical vector, the same size as `name`, with `TRUE` value if the marker file exists in the cache, or `FALSE` otherwise.

`moveFilesIntoCache(src.file.paths, cache.id, name, ext)` :

Moves exisiting files into the cache.

src.file.paths: The current paths of the source files, as a character vector.

cache.id: The cache ID to use.

name: A character vector containing file names.

ext: The extension of the files.

Returned value: None.

`saveContentToFile(content, cache.id, name, ext)` :

Saves content to files into the cache.

content: A list or a character vector containing the contents of the files. It must have the same length as `name`.

cache.id: The cache ID to use.

name: A character vector containing file names.
ext: The extension of the files.
Returned value: None.

`setMarker(cache.id, name)` :
Sets a marker.
cache.id: The cache ID to use.
name: A character vector containing marker names.
Returned value: None.

See Also

BiodbMain.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a compound CSV file database
chebi.tsv <- system.file("extdata", "chebi_extract.tsv", package='biodb')

# Get a connector instance:
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi.tsv)

# Get all entries
entries <- conn$getEntry(conn$getEntryIds())

# Get the cache instance:
cache <- mybiodb$getPersistentCache()

# Get list of files inside the cache:
files <- cache$listFiles(conn$getCacheId())

# Delete files inside the cache:
cache$deleteAllFiles(conn$getCacheId())

# Terminate instance.
mybiodb$terminate()
```

BiodbRemotedbConn-class

The mother class of all remote database connectors.

Description

This is the super class of remote database connectors. It thus defines methods related to remote connection, like the definition of a token, and URL definitions. As with BiodbConn class, you won't need to use the constructor. Nevertheless we provide in the Fields section information about the constructor parameters, for eventual developers.

Methods

`getEntryContentFromDb(entry.id)` :

Get the contents of entries directly from the database. A direct request or an access to the database will be made in order to retrieve the contents. No access to the biodb cache system will be made.

entry.id: A character vector with the IDs of entries to retrieve.

Returned value: A character vector, the same size of entry.id, with contents of the requested entries. An NA value will be set for the content of each entry for which the retrieval failed.

`getEntryContentRequest(entry.id, concatenate = TRUE, max.length = 0)` :

Gets the URL to use in order to get the contents of the specified entries.

entry.id: A character vector with the IDs of entries to retrieve.

concatenate: If set to TRUE, then try to build as few URLs as possible, sending requests with several identifiers at once.

max.length: The maximum length of the URLs to return, in number of characters.

Returned value: A list of BiodbUrl objects.

`getEntryImageUrl(entry.id)` :

Gets the URL to a picture of the entry (e.g.: a picture of the molecule in case of a compound entry).

entry.id: A character vector containing entry IDs.

Returned value: A character vector, the same length as ‘entry.id‘, containing for each entry ID either a URL or NA if no URL exists.

`getEntryPageUrl(entry.id)` :

Gets the URL to the page of the entry on the database web site.

entry.id: A character vector with the IDs of entries to retrieve.

Returned value: A list of BiodbUrl objects, the same length as ‘entry.id‘.

See Also

Super class `BiodbConn` and `BiodbRequestScheduler`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Define ChEBI connector example
defFile <- system.file("extdata", "chebi_ex.yml", package="biodb")
connFile <- system.file("extdata", "ChebiExConn.R", package="biodb")
entryFile <- system.file("extdata", "ChebiExEntry.R", package="biodb")
mybiodb$loadDefinitions(defFile)
source(connFile)
source(entryFile)

# Get connector
conn <- mybiodb$getFactory()$createConn('chebi.ex')

# Get the picture URL of an entry
```

```
picture.url <- conn$getEntryImageUrl('15440')

# Get the page URL of an entry
page.url <- conn$getEntryPageUrl('15440')

# Terminate instance.
mybiodb$terminate()
```

BiodbRequest *Class Request.*

Description

Class Request.

Class Request.

Details

This class represents a Request object that can be used with the Request Scheduler.

Methods

Public methods:

- BiodbRequest$new()
- BiodbRequest$setConn()
- BiodbRequest$getConn()
- BiodbRequest$getUrl()
- BiodbRequest$getMethod()
- BiodbRequest$getEncoding()
- BiodbRequest$getCurlOptions()
- BiodbRequest$getUniqueKey()
- BiodbRequest$getHeaderAsSingleString()
- BiodbRequest$getBody()
- BiodbRequest$print()
- BiodbRequest$toString()
- BiodbRequest$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbRequest$new(
  url,
  method = c("get", "post"),
  header = character(),
  body = character(),
```

```
  encoding = integer(),
  conn = NULL
)
```

Arguments:

`url` A `BiodbUrl` object.

`method` HTTP method. Either "get" or "post".

`header` The header.

`body` The body.

`encoding` The encoding to use.

`conn` A valid BiodbConn instance for which this request is built.

Returns: A new instance.

Method `setConn()`**:** Sets the associated connector (usually the connector that created this request).

Usage:

```
BiodbRequest$setConn(conn)
```

Arguments:

`conn` A valid BiodbConn object.

Returns: None.

Method `getConn()`**:** gets the associated connector (usually the connector that created this request).

Usage:

```
BiodbRequest$getConn()
```

Returns: The associated connector as a BiodbConn object.

Method `getUrl()`**:** Gets the URL.

Usage:

```
BiodbRequest$getUrl()
```

Returns: The URL as a BiodbUrl object.

Method `getMethod()`**:** Gets the method.

Usage:

```
BiodbRequest$getMethod()
```

Returns: The method as a character value.

Method `getEncoding()`**:** Gets the encoding.

Usage:

```
BiodbRequest$getEncoding()
```

Returns: The encoding.

Method `getCurlOptions()`**:** Gets the options object to pass to cURL library.

Usage:

```
BiodbRequest$getCurlOptions(useragent)
```

Arguments:

`useragent` The user agent as a character value.

Returns: An RCurl options object.

Method `getUniqueKey()`**:** Gets a unique key to identify this request. The key is an MD5 sum computed from the string representation of this request.

Usage:

```
BiodbRequest$getUniqueKey()
```

Returns: A unique key as an MD5 sum.

Method `getHeaderAsSingleString()`**:** Gets the HTTP header as a string, concatenating all its information into a single string.

Usage:

```
BiodbRequest$getHeaderAsSingleString()
```

Returns: The header as a single character value.

Method `getBody()`**:** Gets the body.

Usage:

```
BiodbRequest$getBody()
```

Returns: The body as a character value.

Method `print()`**:** Displays information about this instance.

Usage:

```
BiodbRequest$print()
```

Returns: None.

Method `toString()`**:** Gets a string representation of this instance.

Usage:

```
BiodbRequest$toString()
```

Returns: A single string giving a representation of this instance.

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

Usage:

```
BiodbRequest$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BiodbRequestScheduler`, `BiodbUrl`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Create a request object
u <- 'https://www.ebi.ac.uk/webservices/chebi/2.0/test/getCompleteEntity'
url <- BiodbUrl$new(url=u)
url$setParam('chebiId', 15440)
request <- BiodbRequest$new(method='get', url=url)

# Send request
mybiodb$getRequestScheduler()$sendRequest(request)

# Terminate instance.
mybiodb$terminate()
```

BiodbRequestScheduler-class

Class for handling requests.

Description

This class handles GET and POST requests, as well as file downloading. Each remote database connection instance (instance of concrete class inheriting from `BiodbRemotedbConn`, creates an instance of `BiodbRequestScheduler` for handling database connection. A timer is used to schedule connections, and avoid sending too much requests to the database. This class is not meant to be used directly by the library user. See section Fields for a list of the constructor's parameters.

Methods

`downloadFile(url, dest.file)` :

Downloads the content of a URL and save it into the specified destination file.

url: The URL to access, as a BiodbUrl object.

dest.file: A path to a destination file.

Returned value: None.

`getUrl( url, params = list(), method = c("get", "post"), header = character(), body = character(), encoding`

Send a URL request, either with GET or POST method, and return result.

`getUrlString(url, params = list())` Build a URL string, using a base URL and parameters to be passed.

`sendRequest(request, cache.read = TRUE)` :

Sends a request, and returns content result.

request: A BiodbRequest instance.

cache.read: If set to TRUE, the cache system will be used. In case the same request has already been already run and its results saved into the cache, then the request is not run again, the targeted server not contacted, and the results are directly loaded from the cache system.

Returned value: The results returned by the contacted server, as a single string value.

`sendSoapRequest( url, soap.request, soap.action = NA_character_, encoding = integer() )`
:
Sends a SOAP request to a URL. Returns the string result.
url: The URL to access, as a character string.
soap.request: The XML SOAP request to send, as a character string.
soap.action: The SOAP action to contact, as a character string.
encoding: The encoding to use.
Returned value: The results returned by the contacted server, as a single string value.

See Also

BiodbRemotedbConn, BiodbRequestSchedulerRule, BiodbConnObserver.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get the scheduler
sched <- mybiodb$getRequestScheduler()

# Create a request object
u <- 'https://www.ebi.ac.uk/webservices/chebi/2.0/test/getCompleteEntity'
url <- BiodbUrl$new(url=u)
url$setParam('chebiId', 15440)
request <- BiodbRequest$new(method='get', url=url)

# Send request
sched$sendRequest(request)

# Terminate instance.
mybiodb$terminate()
mybiodb <- NULL
```

BiodbRequestSchedulerRule

Scheduler rule class.

Description

Scheduler rule class.

Scheduler rule class.

Details

This class represents a rule for the request scheduler.

Methods

Public methods:

Method `new()`**:** Constructor.

Usage:

```
BiodbRequestSchedulerRule$new(host, conn = NULL)
```

Arguments:

`host` The web host for which this rules is applicable.

`conn` The connector instance that is concerned by this rule.

Method `getHost()`**:** Gets host.

Usage:

```
BiodbRequestSchedulerRule$getHost()
```

Returns: Returns the host.

Method `getN()`**:** Gets N value. The number of connections allowed during a period of T seconds.

Usage:

```
BiodbRequestSchedulerRule$getN()
```

Returns: Returns N as an integer.

Method `getT()`**:** Gets T value. The number of seconds during which N connections are allowed.

Usage:

```
BiodbRequestSchedulerRule$getT()
```

Returns: Returns T as a numeric.

Method `setFrequency()`**:** Sets both N and T.

Usage:

```
BiodbRequestSchedulerRule$setFrequency(n, t)
```

Arguments:

`n` The number of connections allowed during a period of t seconds, as an integer.

`t` The number of seconds during which n connections are allowed, as a numeric value.

Returns: None.

Method `getConnectors()`**:** Gets connectors associaated with this rule.

Usage:

```
BiodbRequestSchedulerRule$getConnectors()
```

Returns: A list of BiodbConn objects.

Method `addConnector()`**:** Associate a connector with this rule.

Usage:

```
BiodbRequestSchedulerRule$addConnector(conn)
```

Arguments:

`conn` A BiodbConn object.

Returns: None.

Method `removeConnector()`**:** Disassociate a connector from this rule.

Usage:

```
BiodbRequestSchedulerRule$removeConnector(conn)
```

Arguments:

`conn` A BiodbConn instance.

Returns: None.

Method `print()`**:** Displays information about this instance.

Usage:

```
BiodbRequestSchedulerRule$print()
```

Returns: None.

Method `waitAsNeeded()`**:** Wait (sleep) until a new request is allowed.

Usage:

```
BiodbRequestSchedulerRule$waitAsNeeded()
```

Returns: Nothing.

Method `recomputeFrequency()`**:** Recompute frequency from submitted N and T values.

Usage:

```
BiodbRequestSchedulerRule$recomputeFrequency()
```

Returns: Nothing.

Method `computeSleepTime()`**:** Compute the needed sleep time to wait until a new request is allowed, starting from the submitted time.

Usage:

```
BiodbRequestSchedulerRule$computeSleepTime(cur.time = Sys.time())
```

Arguments:

`cur.time` Time from which to compute needed sleep time.

Returns: The needed sleep time in seconds.

Method `storeCurrentTime()`**:** Stores the current time.

Usage:

```
BiodbRequestSchedulerRule$storeCurrentTime(cur.time = Sys.time())
```

Arguments:

`cur.time` The current time.

Returns: Nothing.

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

Usage:

```
BiodbRequestSchedulerRule$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BiodbRequestScheduler`.

`BiodbSdfEntry-class` *Entry class for content in SDF format.*

Description

This is an abstract class for handling database entries whose content is in SDF format.

See Also

Super class `BiodbTxtEntry`.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbSdfEntry")
```

BiodbSqlBinaryOp *This class represents an SQL binary operator.*

Description

This class represents an SQL binary operator.

This class represents an SQL binary operator.

Super class

biodb::BiodbSqlExpr -> BiodbSqlBinaryOp

Methods

Public methods:

- BiodbSqlBinaryOp$new()
- BiodbSqlBinaryOp$toString()
- BiodbSqlBinaryOp$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbSqlBinaryOp$new(lexpr, op, rexpr)
```

Arguments:

`lexpr` A BiodbSqlExpr instance for the left part.

`op` The binary operator, as a string.

`rexpr` A BiodbSqlExpr instance for the right part.

Returns: A new instance.

Method `toString()`**:** Converts into a string.

Usage:

```
BiodbSqlBinaryOp$toString()
```

Returns: A string containing the SQL expression.

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

Usage:

```
BiodbSqlBinaryOp$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

BiodbSqlExpr *The SQL Expression abstact class.*

Description

The SQL Expression abstact class.

The SQL Expression abstact class.

Details

This abstract class represents an SQL expression.

Methods

Public methods:

- `BiodbSqlExpr$toString()`
- `BiodbSqlExpr$clone()`

Method `toString()`**:** Converts into a string.

Usage:

```
BiodbSqlExpr$toString()
```

Returns: A string containing the SQL expression.

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

Usage:

```
BiodbSqlExpr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

BiodbSqlField *This class represents an SQL field.*

Description

This class represents an SQL field.

This class represents an SQL field.

Super class

`biodb::BiodbSqlExpr` -> `BiodbSqlField`

Methods

Public methods:

- BiodbSqlField$new()
- BiodbSqlField$toString()
- BiodbSqlField$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbSqlField$new(table = NA_character_, field)
```

Arguments:

`table` The table name.

`field` The field name.

Returns: A new instance.

Method `toString()`**:** Converts into a string.

Usage:

```
BiodbSqlField$toString()
```

Returns: A string containing the SQL expression.

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

Usage:

```
BiodbSqlField$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`BiodbSqlList` *This class represents an SQL list.*

Description

This class represents an SQL list.

This class represents an SQL list.

Super class

biodb::BiodbSqlExpr -> BiodbSqlList

Methods

Public methods:

- BiodbSqlList$new()
- BiodbSqlList$toString()
- BiodbSqlList$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbSqlList$new(values)
```

Arguments:

`values` The values of the list.

Returns: A new instance.

Method `toString()`**:** Converts into a string.

Usage:

```
BiodbSqlList$toString()
```

Returns: A string containing the SQL expression.

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

Usage:

```
BiodbSqlList$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`BiodbSqlLogicalOp` *This class represents an SQL logical operator.*

Description

This class represents an SQL logical operator.

This class represents an SQL logical operator.

Super class

`biodb::BiodbSqlExpr` -> `BiodbSqlLogicalOp`

Methods

Public methods:

- BiodbSqlLogicalOp$new()
- BiodbSqlLogicalOp$addExpr()
- BiodbSqlLogicalOp$toString()
- BiodbSqlLogicalOp$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbSqlLogicalOp$new(op)
```

Arguments:

`op` The logical operator, as a string.

Returns: A new instance.

Method `addExpr()`**:** Add an SQL expression to the logical operator.

Usage:

```
BiodbSqlLogicalOp$addExpr(expr)
```

Arguments:

`expr` A BiodbSqlExpr instance.

Returns: Nothing.

Method `toString()`**:** Converts into a string.

Usage:

```
BiodbSqlLogicalOp$toString()
```

Returns: A string containing the SQL expression.

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

Usage:

```
BiodbSqlLogicalOp$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`BiodbSqlQuery` *This class handles an SQL Query.*

Description

This class handles an SQL Query.

This class handles an SQL Query.

Details

This class represents an SQL query. It is used internally to generate an SQL query string.

Methods

Public methods:

- `BiodbSqlQuery$new()`
- `BiodbSqlQuery$setTable()`
- `BiodbSqlQuery$addField()`
- `BiodbSqlQuery$setDistinct()`
- `BiodbSqlQuery$setLimit()`
- `BiodbSqlQuery$addJoin()`
- `BiodbSqlQuery$setWhere()`
- `BiodbSqlQuery$getJoin()`
- `BiodbSqlQuery$getWhere()`
- `BiodbSqlQuery$getFields()`
- `BiodbSqlQuery$toString()`
- `BiodbSqlQuery$clone()`

Method `new()`**:** Constructor.

Usage:

```
BiodbSqlQuery$new()
```

Returns: A new instance.

Method `setTable()`**:** Set the table.

Usage:

```
BiodbSqlQuery$setTable(table)
```

Arguments:

`table` The table name.

Returns: Nothing.

Method `addField()`**:** Set the fields.

Usage:

```
BiodbSqlQuery$addField(table = NULL, field)
```

Arguments:

`table` The table name.

`field` A field name.

Returns: Nothing.

Method `setDistinct()`**:** Set or unset distinct modifier.

Usage:

```
BiodbSqlQuery$setDistinct(distinct)
```

Arguments:

`distinct` Either TRUE or FALSE for setting or unsetting the distinct flag.

Returns: Nothing.

Method `setLimit()`**:** Set results limit.

Usage:

```
BiodbSqlQuery$setLimit(limit)
```

Arguments:

`limit` The limit to set, as an integer value.

Returns: Nothing.

Method `addJoin()`**:** Add a join.

Usage:

```
BiodbSqlQuery$addJoin(table1, field1, table2, field2)
```

Arguments:

`table1` The first table.

`field1` The field of the first table.

`table2` The second table.

`field2` The field of the second table.

Returns: Nothing.

Method `setWhere()`**:** Set the where clause.

Usage:

```
BiodbSqlQuery$setWhere(expr)
```

Arguments:

`expr` A BiodbSqlExpr representing the "where" clause.

Returns: Nothing.

Method `getJoin()`**:** Builds and returns the join expression.

Usage:

```
BiodbSqlQuery$getJoin()
```

Returns: A character vector representing the join expression.

Method `getWhere()`**:** Gets the where expression.

Usage:

```
BiodbSqlQuery$getWhere()
```

Returns: The BiodbSqlExpr instance representing the "where" clause.

Method `getFields()`**:** Gets the fields to retrieve.

Usage:

```
BiodbSqlQuery$getFields()
```

Returns: A string containing the list of fields to retrieve.

Method `toString()`**:** Generates the string representation of this query.

Usage:

```
BiodbSqlQuery$toString()
```

Returns: A string containing the full SQL query.

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

Usage:

```
BiodbSqlQuery$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BiodbRequestScheduler`, `BiodbRequest`.

`BiodbSqlValue` *This class represents an SQL value.*

Description

This class represents an SQL value.

This class represents an SQL value.

Super class

`biodb::BiodbSqlExpr` -> `BiodbSqlValue`

Methods

Public methods:

- BiodbSqlValue$new()
- BiodbSqlValue$toString()
- BiodbSqlValue$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbSqlValue$new(value)
```

Arguments:

`value` The value.

Returns: A new instance.

Method `toString()`**:** Converts into a string.

Usage:

```
BiodbSqlValue$toString()
```

Returns: A string containing the SQL expression.

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

Usage:

```
BiodbSqlValue$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`BiodbTestMsgAck-class` *A class for acknowledging messages during tests.*

Description

This observer is used to call a testthat::expect_*() method each time a message is received. This is used when running tests on Travis-CI, so Travis does not stop tests because no change is detected in output.

Methods

`notifyProgress(what, index, total)` :

Notify about the progress of an action.

what: A short description of the action.

index: A positive integer number indicating the progress.

total: The maximum for "index". When reached, the action is completed.

Returned value: None.

Examples

```
# To use the acknowledger, set ack=TRUE when creating the Biodb test
# instance:
biodb <- createBiodbTestInstance(ack=TRUE)

# Terminate the BiodbMain instance
biodb$terminate()
```

`BiodbTxtEntry-class` *Entry class for content in text format.*

Description

This is an abstract class for handling database entries whose content is in text format.

See Also

Super class `BiodbEntry`.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbTxtEntry")
```

`BiodbUrl` *Class URL.*

Description

Class URL.

Class URL.

Details

This class represents a URL object that can be used in requests.

Methods

Public methods:

- BiodbUrl$new()
- BiodbUrl$getDomain()
- BiodbUrl$setUrl()
- BiodbUrl$setParam()
- BiodbUrl$print()
- BiodbUrl$toString()
- BiodbUrl$clone()

Method `new()`**:** Constructor.

Usage:

```
BiodbUrl$new(url = character(), params = character())
```

Arguments:

`url` The URL to access, as a character vector.

`params` The list of parameters to append to this URL.

Returns: A new instance.

Method `getDomain()`**:** Gets the domain.

Usage:

```
BiodbUrl$getDomain()
```

Returns: None.

Method `setUrl()`**:** Sets the base URL string.

Usage:

```
BiodbUrl$setUrl(url)
```

Arguments:

`url` The base URL string.

Returns: None.

Method `setParam()`**:** Sets a parameter.

Usage:

```
BiodbUrl$setParam(key, value)
```

Arguments:

`key` The parameter name.

`value` The value of the parameter.

Returns: None.

Method `print()`**:** Displays information about this instance.

Usage:

```
BiodbUrl$print()
```

Returns: None.

Method `toString()`**:** Gets the URL as a string representation.

Usage:

```
BiodbUrl$toString(encode = TRUE)
```

Arguments:

`encode` If set to TRUE, then encodes the URL.

Returns: The URL as a string, with all parameters and values set.

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

Usage:

```
BiodbUrl$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`BiodbRequestScheduler`, `BiodbRequest`.

Examples

```
# Create a URL object
u <- c("https://www.uniprot.org", "uniprot")
p <- c(query="reviewed:yes+AND+organism:9606",
       columns='id,entry name,protein names',
       format="tab")
url <- BiodbUrl$new(url=u, params=p)
url$toString()
```

BiodbWritable-class *An abstract class (more like an interface) to model a writable database.*

Description

A database class that implements this interface must allow the addition of new entries.

Methods

`allowWriting()` :

Allows the connector to write into this database.

Returned value: None.

`disallowWriting()` :

Disallows the connector to write into this database.

Returned value: None.

`setWritingAllowed(allow)` :

Allows or disallows writing for this database.

allow: If set to TRUE, allows writing.

Returned value: None.

`write()` :

Writes into the database. All modifications made to the database since the last time write() was called will be saved.

Returned value: None.

`writingIsAllowed()` :

Tests if the connector has access right to the database.

Returned value: TRUE if writing is allowed for this database, FALSE otherwise.

See Also

BiodbConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Create an empty MASS SQLite database
mydb <- mybiodb$getFactory()$createConn('mass.sqlite')

# Create new entry object
entry <- mybiodb$getFactory()$createNewEntry('mass.sqlite')
entry$setFieldValue('accession', '0')
entry$setFieldValue('name', 'Some Entry')

# Add the new entry
mydb$allowEditing()
mydb$addNewEntry(entry)

# Write the database
mydb$allowWriting()
mydb$setUrl('base.url', 'mydatabase.sqlite')
mydb$write()

# Terminate instance.
mybiodb$terminate()
mybiodb <- NULL
```

BiodbXmlEntry-class *Entry class for content in XML format.*

Description

This is an abstract class for handling database entries whose content is in XML format.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- methods::setRefClass("MyEntry", contains="BiodbXmlEntry")
```

---

closeMatchPpm *Close match PPM*

Description

Matches peaks between two spectra.

Usage

```
closeMatchPpm(x, y, xidx, yidx, xolength, dppm, dmz)
```

Arguments

x	sorted M/Z values (ascending order) of input spectrum (no NA).
y	sorted M/Z values (ascending order) of reference spectrum (no NA).
xidx	indices of the M/Z peaks of x, taken from the original spectrum ordered in decreasing intensity values.
yidx	indices of the M/Z peaks of y, taken from the original spectrum ordered in decreasing intensity values.
xolength	The length of the output.
dppm	The M/Z tolerance in PPM.
dmz	Minimum M/Z tolerance.

Value

A list of results.

CompCsvFileConn-class *Compound CSV File connector class.*

Description

This is the connector class for a Compound CSV file database.

See Also

Super class CsvFileConn and interfaces BiodbCompounddbConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a connector:
chebi_file <- system.file("extdata", "chebi_extract.tsv", package="biodb")
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi_file)

# Get an entry
e <- conn$getEntry('')

# Terminate instance.
mybiodb$terminate()
```

CompCsvFileEntry-class *Compound CSV File entry class.*

Description

This is the entry class for Compound CSV file databases.

See Also

Super class BiodbCsvEntry.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a connector that inherits from CsvFileConn:
chebi_file <- system.file("extdata", "chebi_extract.tsv", package="biodb")
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi_file)
```

```
# Get an entry
e <- conn$getEntry('')

# Terminate instance.
mybiodb$terminate()
```

`CompSqliteConn-class` *Class for handling a Compound database in SQLite format.*

Description

This is the connector class for a Compound database.

See Also

Super class `SqliteConn` and interfaces `BiodbCompounddbConn`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a connector:
chebi_file <- system.file("extdata", "chebi_extract.sqlite", package="biodb")
conn <- mybiodb$getFactory()$createConn('comp.sqlite', url=chebi_file)

# Get an entry
e <- conn$getEntry('1018')

# Terminate instance.
mybiodb$terminate()
```

`CompSqliteEntry-class` *Compound SQLite entry class.*

Description

This is the entry class for a Compound SQLite database.

See Also

Super class `BiodbListEntry`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get path to LCMS database example file
lcmsdb <- system.file("extdata", "chebi_extract.sqlite", package="biodb")

# Create a connector
conn <- mybiodb$getFactory()$createConn('comp.sqlite', url=lcmsdb)

# Get an entry
e <- conn$getEntry('34.pos.col12.0.78')

# Terminate instance.
mybiodb$terminate()
```

---

connNameToClassPrefix *Convert connector name into class prefix.*

Description

Converts the connector name into the class prefix (e.g.: "mass.csv.file" –> "MassCsvFile").

Usage

```
connNameToClassPrefix(connName)
```

Arguments

connName	A connector name (e.g.: "mass.csv.file").

Value

The corresponding class prefix (e.g.: "MassCsvFile").

---

createBiodbTestInstance *Creating a BiodbMain instance for tests.*

Description

Creates a BiodbMain instance with options specially adapted for tests. You can request the logging of all messages into a log file. It is also possible to ask for the creation of a BiodbTestMsgAck observer, which will receive all messages and emit a testthat test for each message. This will allow the testthat output to not stall a long time while, for example, downloading or extracting a database. Do not forget to call `terminate()` on your instance at the end of your tests.

Usage

```
createBiodbTestInstance(ack = FALSE)
```

Arguments

ack	If set to TRUE, an instance of BiodbTestMsgAck will be attached to the BiodbMain instance.

Value

The created BiodbMain instance.

Examples

```
# Instantiate a BiodbMain instance for testing
biodb <- biodb::createBiodbTestInstance()

# Terminate the instance
biodb$terminate()
```

CsvFileConn-class *CSV File connector class.*

Description

This is the abstract connector class for all CSV file databases.

Methods

`addField(field, value)` :

Adds a new field to the database. The field must not already exist. The same single value will be set to all entries for this field. A new column will be written in the memory data frame, containing the value given.

field: A valid Biodb entry field name.

value: The value to set for this field.

Returned value: None.

`getCsvQuote()` :

Gets the characters used to delimit quotes in the CSV database file.

Returned value: The characters used to delimit quotes as a single character value.

`getCsvSep()` :

Gets the current CSV separator used for the database file.

Returned value: The CSV separator as a character value.

`getEntryContentFromDb(entry.id)` :

Get the contents of entries directly from the database. A direct request or an access to the database will be made in order to retrieve the contents. No access to the biodb cache system will be made.

entry.id: A character vector with the IDs of entries to retrieve.

Returned value: A character vector, the same size of entry.id, with contents of the requested entries. An NA value will be set for the content of each entry for which the retrieval failed.

`getFieldColName(field)` :

Get the column name corresponding to a Biodb field.

field: A valid Biodb entry field name. This field must be defined for this database instance.

Returned value: The column name from the CSV file.

`getFieldNames()` :

Get the list of all biodb fields handled by this database.

Returned value: A character vector of the biodb field names.

`getNbEntries(count = FALSE)` :

Get the number of entries contained in this database.

count: If set to TRUE and no straightforward way exists to get number of entries, count the output of getEntryIds().

Returned value: The number of entries in the database, as an integer.

`hasField(field)` :

Tests if a field is defined for this database instance.

field: A valid Biodb entry field name.

Returned value: TRUE of the field is defined, FALSE otherwise.

`isSearchableByField(field)` :

Tests if a field can be used to search entries when using methods searchByName() and searchCompound().

field: The name of the field.

Returned value: Returns TRUE if the database is searchable using the specified field, FALSE otherwise.

`setCsvQuote(quote)` :

Sets the characters used to delimit quotes in the CSV database file.

quote: The characters used to delimit quotes as a single character value. You may specify several characters. Example: "\"'".

Returned value: None.

`setCsvSep(sep)` :

Sets the CSV separator to be used for the database file. If this method is called after the loading of the database, it will throw an error.

sep: The CSV separator as a character value.

Returned value: None.

`setDb(db)` :

Sets the database directly from a data frame. You must not have set the database previously with the URL parameter.

db: A data frame containing your database.

Returned value: None.

`setField(field, colname, ignore.if.missing = FALSE)` :

Sets a field by making a correspondence between a Biodb field and one or more columns of the loaded data frame.

field: A valid Biodb entry field name. This field must not be already defined for this database instance.

colname: A character vector containing one or more column names from the CSV file.

ignore.if.missing: Deprecated parameter.

Returned value: None.

`show()` :

Prints a description of this connector.

Returned value: None.

See Also

Super classes BiodbConn, BiodbWritable, BiodbEditable, and sub-classes CompCsvFileConn, MassCsvFileConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a connector that inherits from CsvFileConn:
chebi_file <- system.file("extdata", "chebi_extract.tsv", package="biodb")
conn <- mybiodb$getFactory()$createConn('comp.csv.file', url=chebi_file)

# Get an entry
e <- conn$getEntry('1018')

# Terminate instance.
mybiodb$terminate()
```

---

df2str *Convert a data.frame into a string.*

Description

Prints a data frame (partially if too big) into a string.

Usage

```
df2str(x, rowCut = 5, colCut = 5)
```

Arguments

x	The data frame object.
rowCut	The maximum of rows to print.
colCut	The maximum of columns to print.

Value

A string containing the data frame representation (or part of it).

Examples

```
# Converts the first 5 rows and first 6 columns of a data frame into a
# string:
x <- data.frame(matrix(1:160, nrow=10, byrow=TRUE))
s <- df2str(x, rowCut=5, colCut=6)
```

---

error *Throw an error and log it too.*

Description

Throws am error and logs it too with biodb logger.

Usage

```
error(...)
```

Arguments

...	Values to be passed to sprintf().

Value

Nothing.

Examples

```
# Throws an error:
tryCatch(biodb::error('Index is %d.', 10), error=function(e){e$message})
```

error0 *Throw an error and log it too.*

Description

Throws an error and logs it too with biodb logger, using paste0().

Usage

```
error0(...)
```

Arguments

...	Values to be passed to paste0().

Value

Nothing.

Examples

```
# Throws an error:
tryCatch(biodb::error0('Index is ', 10, '.'), error=function(e){e$message})
```

ExtConnClass *Extension connector clas*

Description

A class for generating a new connector class.

Details

This class generates a new connector class from given parameters. The new class can inherit directly from BiodbConn or BiodbCompounddbConn or BiodbMassdbConn. It can also be editable and/or writable.

Super classes

biodb::ExtGenerator -> biodb::ExtFileGenerator -> ExtConnClass

Methods

Public methods:

- ExtConnClass$new()
- ExtConnClass$clone()

Method `new()`**:** Constructor

Usage:

```
ExtConnClass$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtConnClass$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new connector class inside R folder:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtConnClass$new(path=pkgFolder, dbName='foo.db',
                        dbTitle='Foo database',
                        connType='mass', remote=TRUE)$generate()
```

`ExtCpp` *Extension C++ code class*

Description

A class for generating C++ example files (code & test).

Details

This class generates examples of an R function written in C++ using Rcpp, of a pure C++ function used to speed up computing, and of C++ code for testing the pure C++ function. As for the R function written with Rcpp, it is tested inside standard testthat R code.

Super class

`biodb::ExtGenerator` -> `ExtCpp`

Methods

Public methods:

- ExtCpp$new()
- ExtCpp$clone()

Method `new()`**:** Constructor

Usage:

```
ExtCpp$new(...)
```

Arguments:

`...` See the constructor of ExtGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtCpp$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate C++ files
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtCpp$new(path=pkgFolder)$generate()
```

`ExtDefinitions` *Extension defintions file class*

Description

A class for generating the definitions.yml file of a new extension package.

Details

This class generates the definitions.yml of a new extension package, needed for defininig the new connector.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtDefinitions`

Methods

Public methods:

- ExtDefinitions$new()
- ExtDefinitions$clone()

Method `new()`**:** Constructor

Usage:

```
ExtDefinitions$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters. offers this possiblity.

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

Usage:

```
ExtDefinitions$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate the biodb definitions.yml file inside "inst" folder:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtDefinitions$new(path=pkgFolder, dbName='foo.db',
                          dbTitle='Foo database')$generate()
```

`ExtDescriptionFile` *Extension DESCRIPTION file*

Description

A class for generating a DESCRIPTION file for an extension package.

Details

This class generates a DESCRIPTION for a biodb extension package.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtDescriptionFile`

Methods

Public methods:

- ExtDescriptionFile$new()
- ExtDescriptionFile$clone()

Method `new()`**:** Constructor.

Usage:

```
ExtDescriptionFile$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtDescriptionFile$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate the DESCRIPTION file:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtDescriptionFile$new(path=pkgFolder, dbName='foo.db',
                              dbTitle='Foo database', email='j.smith@e.mail',
                              firstname='John', lastname='Smith', rcpp=TRUE,
                              entryType='xml')$generate()
```

ExtEntryClass *Extension entry class*

Description

A class for generating a new entry class.

Details

This class generates a new entry class from given parameters. The new class can inherit directly from `BiodbEntry` or from one of its sub-classes: `BiodbCsvEntry`, `BiodbHtmlEntry`, ...

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtEntryClass`

Methods

Public methods:

- ExtEntryClass$new()
- ExtEntryClass$clone()

Method `new()`**:** Constructor

Usage:

```
ExtEntryClass$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtEntryClass$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new entry class inside R folder:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtEntryClass$new(path=pkgFolder, dbName='foo.db',
                         dbTitle='Foo database',
                         connType='mass', entryType='xml')$generate()
```

`ExtFileGenerator` *Extension file generator abstract class*

Description

The mother class of all file generators for biodb extension packages.

Details

All file generator classes for biodb extensions must inherit from this class.

Super class

`biodb::ExtGenerator` -> `ExtFileGenerator`

Methods

Public methods:

- ExtFileGenerator$new()
- ExtFileGenerator$clone()

Method `new()`**:** Constructor

Usage:

```
ExtFileGenerator$new(
  filename = NULL,
  overwrite = FALSE,
  folder = character(),
  template = NULL,
  upgrader = c("fullReplacer", "lineAdder"),
  ...
)
```

Arguments:

`filename` The name of the generated file.

`overwrite` If set to TRUE, then overwrite existing destination file, even whatever the version of the template file. If set to FALSE, only overwrite if the version of the template file is strictly greater than the existing destination file.

`folder` The destination subfolder inside the package directory, as a character vector of subfolders hierarchy.

`template` The filename of the template to use.

`upgrader` The type of upgrader to use. "fullReplacer" replaces the whole destination file by the template if it is newer (it compares version numbers). "lineAdder" only adds to the destination file the missing lines from the template file.

`...` See the constructor of ExtGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtFileGenerator$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtConnClass$new(path=pkgFolder, dbName='foo.db',
                        dbTitle='Foo database',
                        connType='mass', remote=TRUE)$generate()
```

`ExtGenerator` *Extension generator abstract class*

Description

The mother class of all generators for biodb extension packages.

Details

All generator classes for biodb extensions must inherit from this class.

Methods

Public methods:

- `ExtGenerator$new()`
- `ExtGenerator$generate()`
- `ExtGenerator$upgrade()`
- `ExtGenerator$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtGenerator$new(
  path,
  loadCfg = TRUE,
  saveCfg = TRUE,
  pkgName = getPkgName(path),
  email = "author@e.mail",
  dbName = "foo.db",
  dbTitle = "Foo database",
  pkgLicense = getLicenses(),
  firstname = "Firstname of author",
  lastname = "Lastname of author",
  newPkg = FALSE,
  connType = getConnTypes(),
  entryType = getEntryTypes(),
  editable = FALSE,
  writable = FALSE,
  remote = FALSE,
  downloadable = FALSE,
  makefile = FALSE,
  rcpp = FALSE,
  vignetteName = "intro",
  githubRepos = getReposName(path, default = "myaccount/myrepos")
)
```

Arguments:

`path` The path to the package folder.

`loadCfg` Set to FALSE to disable loading of tag values from config file "biodb_ext.yml".

`saveCfg` Set to FALSE to disable saving of tag values into config file "biodb_ext.yml".

`pkgName` The package name. If set to NULL, the folder name pointer by the "path" paramater will be used as the package name.

`email` The email of the author.

`dbName` The name of the database (in biodb format "my.db.name"), that will be used in "definitions.yml" file and for connector and entry classes.

`dbTitle` The official name of the database (e.g.: HMDB, UniProtKB, KEGG).

`pkgLicense` The license of the package.

`firstname` The firstname of the author.

`lastname` The lastname of the author.

`newPkg` Set to TRUE if the package is not yet published on Bioconductor.

`connType` The type of connector class to implement.

`entryType` The type of entry class to implement.

`editable` Set to TRUE to allow the generated connector to create new entries in memory.

`writable` Set to TRUE to enable the generated connector to write into the database.

`remote` Set to TRUE if the database to connect to is not local.

`downloadable` Set to TRUE if the database needs to be downloaded or offers this possiblity.

`makefile` Set to TRUE if you want a Makefile to be generated.

`rcpp` Set to TRUE to enable Rcpp C/C++ code inside the package.

`vignetteName` Set to the name of the default/main vignette.

`githubRepos` Set to the name of the associated GitHub repository. Example: myaccount/myrepos.

Returns: A new instance.

Method `generate()`**:** Generates the destination file(s).

Usage:

```
ExtGenerator$generate(overwrite = FALSE, fail = TRUE)
```

Arguments:

`overwrite` If set to TRUE and destination files exist, overwrite the destination files.

`fail` If set to FALSE, do not fail if destination files exist, just do nothing and return.

Examples:

```
# Generate a new extension package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
biodb::ExtPackage$new(pkgFolder)$generate()
```

Method `upgrade()`**:** Upgrade the destination file(s).

Usage:

```
ExtGenerator$upgrade(generate = TRUE)
```

Arguments:

`generate` If set to FALSE, and destination file(s) do not exist, then do not generate them.

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

Usage:

```
ExtGenerator$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Generate a new connector class inside the R folder:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtConnClass$new(path=pkgFolder, dbName='foo.db',
                        dbTitle='Foo database',
                        connType='mass', remote=TRUE)$generate()


## ------------------------------------------------
## Method `ExtGenerator$generate`
## ------------------------------------------------

# Generate a new extension package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
biodb::ExtPackage$new(pkgFolder)$generate()
```

ExtGitignore *Extension Gitignore file class*

Description

A class for generating the .gitignore file of an extension package.

Details

This class can be used to generate a new .gitignore file or to keep one up to date.

Super classes

biodb::ExtGenerator -> biodb::ExtFileGenerator -> ExtGitignore

Methods

Public methods:

- ExtGitignore$new()
- ExtGitignore$clone()

Method new(): Constructor

Usage:

```
ExtGitignore$new(...)
```

Arguments:

... See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtGitignore$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtGitignore$new(path=pkgFolder)$generate()
```

`ExtLicense` *Extension license*

Description

A class for generating or upgrading the license of a biodb extension package.

Details

This class generates the license for a new extension package, or update the license of an existing one.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtLicense`

Methods

Public methods:

- `ExtLicense$new()`
- `ExtLicense$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtLicense$new(...)
```

Arguments:

... See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtLicense$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new connector class inside R folder:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtLicense$new(path=pkgFolder)$generate()
```

`ExtMakefile` *Extension Makefile*

Description

A class for generating a Makefile for an extension package.

Details

This class generates a Makefile, usable on UNIX-like platforms, for managing a biodb extension package. Targets are automatically generated for running CRAN check, Bioconductor check, test-that tests, compiling, generating documentation, cleaning, etc.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtMakefile`

Methods

Public methods:

- `ExtMakefile$new()`
- `ExtMakefile$clone()`

Method `new()`**:** Constructor.

Usage:

```
ExtMakefile$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtMakefile$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new connector class inside R folder:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtMakefile$new(path=pkgFolder)$generate()
```

`ExtPackage` *Extension package class*

Description

A class for generating the skeleton of a new extension package.

Details

This class manages the files of an extension package.

It can generate all the files of a new extension package: DESCRIPTION, NEWS, README.md, tests, definitons.yml, etc. Optionnaly it also generates other files like: a `.travis.yml` file for Travis-CI, a Makefile for easing development on UNIX-like platforms outside of Rstudio.

It can also upgrade files of an existing package like: definitions.yml, Makefile, .travis.yml, LICENSE, etc.

Super class

`biodb::ExtGenerator` -> `ExtPackage`

Methods

Public methods:

- `ExtPackage$new()`
- `ExtPackage$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtPackage$new(...)
```

Arguments:

`...` See the constructor of ExtGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtPackage$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtPackage$new(path=pkgFolder, dbName='foo.db',
                        dbTitle='Foo database', rcpp=TRUE,
                        connType='mass', entryType='txt', downloadable=TRUE,
                        remote=TRUE)$generate()
```

`ExtPackageFile` *Extension package file class.*

Description

A class for generating the package.R file for a biodb extension.

Details

This class generates the package.R file, writing a reference to the generated skeleton vignette, and possibly including directives for C++ code.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtPackageFile`

Methods

Public methods:

- `ExtPackageFile$new()`
- `ExtPackageFile$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtPackageFile$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtPackageFile$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtPackageFile$new(path=pkgFolder, dbName='foo.db')$generate()
```

`ExtRbuildignore` *Extension Rbuildignore file class*

Description

A class for generating the .Rbuildignore file of an extension package.

Details

This class can be used to generate a new .Rbuildignore file or to keep one up to date.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtRbuildignore`

Methods

Public methods:

- `ExtRbuildignore$new()`
- `ExtRbuildignore$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtRbuildignore$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtRbuildignore$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtRbuildignore$new(path=pkgFolder)$generate()
```

`ExtReadme` *Extension README file class*

Description

A class for generating a README file for a new extension package.

Details

Write a README file inside package directory, using a template file.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtReadme`

Methods

Public methods:

- `ExtReadme$new()`
- `ExtReadme$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtReadme$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtReadme$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtReadme$new(path=pkgFolder, dbName='foo.db',
                     dbTitle='Foo database')$generate()
```

ExtTests *Extension tests class*

Description

A class for generating test files.

Details

This class generates a test file for running biodb generic tests, and a test file containing an example of a custom test for this extension.

Super class

`biodb::ExtGenerator` -> `ExtTests`

Methods

Public methods:

- `ExtTests$new()`
- `ExtTests$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtTests$new(...)
```

Arguments:

`...` See the constructor of ExtGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtTests$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtTests$new(path=pkgFolder, dbName='foo.db', rcpp=TRUE,
                    remote=TRUE)$generate()
```

`ExtTravisFile` *Extension Travis YAML file generator class*

Description

A class for generating a .travis.yml file for a new extension package.

Details

Write a .travis.yml file inside the package directory, using a template file.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtTravisFile`

Methods

Public methods:

- `ExtTravisFile$new()`
- `ExtTravisFile$clone()`

Method `new()`**:** Constructor

Usage:

```
ExtTravisFile$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtTravisFile$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtTravisFile$new(path=pkgFolder, email='myname@e.mail')$generate()
```

`ExtVignette` *Extension vignette class*

Description

A class for generating a vignette example for an extension package.

Details

This class generates a vignette file, serving as example to demonstrate the use of the extension package.

Super classes

`biodb::ExtGenerator` -> `biodb::ExtFileGenerator` -> `ExtVignette`

Methods

Public methods:

- `ExtVignette$new()`
- `ExtVignette$clone()`

Method `new()`**:** Constructor.

Usage:

```
ExtVignette$new(...)
```

Arguments:

`...` See the constructor of ExtFileGenerator for the parameters.

Returns: A new instance.

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

Usage:

```
ExtVignette$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::ExtVignette$new(path=pkgFolder, dbName='foo.db',
                       dbTitle='Foo database', vignetteName='main',
                       firstname='John', lastname='Smith',
                       remote=TRUE)$generate()
```

`FileTemplate` *File template class.*

Description

A class for reading a file template, replacing tags inside, and writing the results in an output file.

Methods

Public methods:

- `FileTemplate$new()`
- `FileTemplate$replace()`
- `FileTemplate$choose()`
- `FileTemplate$select()`
- `FileTemplate$write()`
- `FileTemplate$getLines()`
- `FileTemplate$clone()`

Method `new()`**:** Constructor

Usage:

```
FileTemplate$new(path)
```

Arguments:

`path` The path to the template file.

Returns: A new instance.

Method `replace()`**:** Replace a tag by its value inside the template file.

Usage:

```
FileTemplate$replace(tag, value)
```

Arguments:

`tag` The tag to replace.
`value` The value to replace the tag with.

Returns: invisible(self) for chaining method calls.

Method `choose()`**:** Choose one case among a set of cases.

Usage:

```
FileTemplate$choose(set, case)
```

Arguments:

`set` The name of the case set.

`case` The name of case.

Returns: invisible(self) for chaining method calls.

Method `select()`**:** Select or remove sections that match a name.

Usage:

```
FileTemplate$select(section, enable)
```

Arguments:

`section` The name of the section.

`enable` Set to TRUE to select the section (and keep it), and FALSE to remove it.

Returns: invisible(self) for chaining method calls.

Method `write()`**:** Write template with replaced values to disk.

Usage:

```
FileTemplate$write(path, overwrite = FALSE, checkRemainingTags = TRUE)
```

Arguments:

`path` Path to output file.

`overwrite` If set to FALSE and the destination file already exists, a message is thrown. Otherwise writes into the destination.

`checkRemainingTags` If set to TRUE, checks first, before writing, if there any remaining tags that have not been processed. A warning is thrown for each found tag.

Returns: None

Method `getLines()`**:** Get the lines of the templates.

Usage:

```
FileTemplate$getLines()
```

Returns: A vector containing the lines of the templates.

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

Usage:

```
FileTemplate$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

genNewExtPkg *Generate a new extension package for biodb.*

Description

Generates all the necessary files for a new extension package.

Usage

```
genNewExtPkg(...)
```

Arguments

...	Parameters passed to ExtPackage constructor.

Value

Nothing.

See Also

ExtPackage.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::genNewExtPkg(path=pkgFolder, dbName='foo.db',
                    dbTitle='Foo database', rcpp=TRUE,
                    connType='mass', entryType='txt', downloadable=TRUE,
                    remote=TRUE)
```

getConnClassName *Get connector class name.*

Description

Gets the name of the connector class corresponding to a connector.

Usage

```
getConnClassName(connName)
```

Arguments

connName	A connector name (e.g.: "mass.csv.file").

Value

The name of the corresponding connector class (e.g.: "MassCsvFileConn").

Examples

```
biodb::getConnClassName('foo.db')
```

---

getConnTypes *Get connector types.*

Description

Get the list of available connector types.

Usage

```
getConnTypes()
```

Value

A character vector containing the connector types.

Examples

```
biodb::getConnTypes()
```

---

getEntryClassName *Get entry class name.*

Description

Gets the name of the entry class corresponding to a connector.

Usage

```
getEntryClassName(connName)
```

Arguments

connName	A connector name (e.g.: "mass.csv.file").

Value

The name of the corresponding entry class (e.g.: "MassCsvFileEntry").

Examples

```
biodb::getEntryClassName('foo.db')
```

`getEntryTypes` *Get entry types.*

Description

Get the list of available entry types.

Usage

```
getEntryTypes()
```

Value

A character vector containing the entry types.

Examples

```
biodb::getEntryTypes()
```

`getLicenses` *Get the available licenses for extension packages.*

Description

Get the available licenses for extension packages.

Usage

```
getLicenses()
```

Value

A character vector containing license names.

Examples

```
biodb::getLicenses()
```

getLogger *Get the main package logger.*

Description

Gets the main package logger, parent of all loggers of this package.

Usage

```
getLogger()
```

Value

The main package logger (named "biodb") as a lgr::Logger object.

Examples

```
biodb::getLogger()
```

getPkgName *Get the package name from a package folder path.*

Description

The package name is extracted from the path by taking the basename.

Usage

```
getPkgName(pkgRoot, check = TRUE)
```

Arguments

pkgRoot	The path to the root folder of the package.
check	If set to TRUE the extracted package name is checked against regular expression "^biodb[A-Z][A-Za-z0-9]+$", to ensure the format is respected.

Value

The package name of the biodb extension.

Examples

```
biodb::getPkgName('/my/path/to/my/extension/biodbFoo')
```

getReposName *Extract the repository name from a package folder.*

Description

Given the root path of a package, returns the GitHub repository name.

Usage

```
getReposName(pkgRoot, default = NULL)
```

Arguments

pkgRoot	The path to the root folder of the package.
default	A default value to return in case git4r package is not available or the folder is not a Git repository.

Value

The repository name.

Examples

```
biodb::getReposName('/my/path/to/my/extension/biodbFoo')
```

getTestOutputDir *Get the test output directory.*

Description

Returns the path to the test output directory. The function creates this also this directory if it does not exist.

Usage

```
getTestOutputDir()
```

Value

The path to the test output directory, as a character value.

Examples

```
# Get the test output directory:
biodb::getTestOutputDir()
```

listTestRefEntries *List test reference entries.*

Description

Lists the reference entries in the test folder for a specified connector. The test reference files must be in <pkg>/tests/testthat/res/ folder and their names must match entry-<database_name>-<entry_accession>.json (e.g.: entry-comp.csv.file-1018.json).

Usage

```
listTestRefEntries(conn.id, limit = 0)
```

Arguments

conn.id	A valid Biodb connector ID.
limit	The maximum number of entries to retrieve.

Value

A list of entry IDs.

Examples

```
# List IDs of test reference entries:
biodb::listTestRefEntries('comp.csv.file')
```

logDebug *Log debug message.*

Description

Logs a debug level message with biodb logger.

Usage

```
logDebug(...)
```

Arguments

...	Values to be passed to sprintf().

Value

Nothing.

Examples

```
# Logs a debug message:
biodb::logDebug('Index is %d.', 10)
```

`logDebug0` *Log debug message.*

Description

Logs a debug level message with biodb logger, using paste0().

Usage

```
logDebug0(...)
```

Arguments

`...`	Values to be passed to paste0()

Value

Nothing.

Examples

```
# Logs a debug message:
biodb::logDebug0('Index is ', 10, '.')
```

`logInfo` *Log information message.*

Description

Logs an information level message with biodb logger.

Usage

```
logInfo(...)
```

Arguments

`...`	Values to be passed to sprintf().

Value

Nothing.

Examples

```
# Logs an info message:
biodb::logInfo('Index is %d.', 10)
```

`logInfo0` *Log information message.*

Description

Logs an information level message with biodb logger, using paste0().

Usage

```
logInfo0(...)
```

Arguments

`...`	Values to be passed to paste0().

Value

Nothing.

Examples

```
# Logs an info message:
biodb::logInfo0('Index is ', 10, '.')
```

`logTrace` *Log trace message.*

Description

Logs a trace level message with biodb logger.

Usage

```
logTrace(...)
```

Arguments

...	Values to be passed to sprintf().

Value

Nothing.

Examples

```
# Logs a trace message:
biodb::logTrace('Index is %d.', 10)
```

---

logTrace0 *Log trace message.*

Description

Logs a trace level message with biodb logger, using paste0().

Usage

```
logTrace0(...)
```

Arguments

...	Values to be passed to paste0()

Value

Nothing.

Examples

```
# Logs a trace message:
biodb::logTrace0('Index is ', 10, '.')
```

lst2str *Convert a list into a string.*

Description

Prints a string (partially if too big) into a string.

Usage

```
lst2str(x, nCut = 10)
```

Arguments

x	The list to convert into a string.
nCut	The maximum of elements to print.

Value

A string containing the list representation (or part of it).

Examples

```
# Converts the first 5 elements of a list into a string:
s <- lst2str(1:10, nCut=5)
```

MassCsvFileConn-class *Mass CSV File connector class.*

Description

This is the connector class for a MASS CSV file database.

Methods

`addPrecursorFormulae(formulae)` :

Adds new formulae to the list of formulae used to recognize precursors.

formulae: A character vector containing formulae.

Returned value: None.

`getChromCol(ids = NULL)` :

Gets a list of chromatographic columns contained in this database.

ids: A character vector of entry identifiers (i.e.: accession numbers). Used to restrict the set of entries on which to run the algorithm.

Returned value : A data.frame with two columns, one for the ID 'id' and another one for the title 'title'.

`getNbPeaks(mode = NULL, ids = NULL)` :

Gets the number of peaks contained in the database.

mode: The MS mode. Set it to either 'neg' or 'pos' to limit the counting to one mode.

ids: A character vector of entry identifiers (i.e.: accession numbers). Used to restrict the set of entries on which to run the algorithm.

Returned value: The number of peaks, as an integer.

`getPrecursorFormulae()` :

Gets the list of formulae used to recognize precursors.

Returned value: A character vector containing chemical formulae.

`isAPrecursorFormula(formula)` :

Tests if a formula is a precursor formula.

formula: A chemical formula, as a character value.

Returned value: TRUE if the submitted formula is considered a precursor.

`setPrecursorFormulae(formulae)` :

Sets the list precursor formulae.

formulae: A character vector containing formulae.

Returned value: None.

See Also

Super class `CsvFileConn` and interfaces `BiodbMassdbConn`, `BiodbWritable` and `BiodbEditable`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get path to LCMS database example file
lcmsdb <- system.file("extdata",
                      "massbank_extract_lcms_2.tsv", package="biodb")

# Create a connector
conn <- mybiodb$getFactory()$createConn('mass.csv.file', url=lcmsdb)

# Get an entry
e <- conn$getEntry('PR010001')

# Terminate instance.
mybiodb$terminate()
```

MassCsvFileEntry-class *Mass CSV File entry class.*

Description

This is the entry class for Mass CSV file databases.

See Also

Super class BiodbCsvEntry.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get path to LCMS database example file
lcmsdb <- system.file("extdata",
                      "massbank_extract_lcms_2.tsv", package="biodb")

# Create a connector
conn <- mybiodb$getFactory()$createConn('mass.csv.file', url=lcmsdb)

# Get an entry
e <- conn$getEntry('PR010001')

# Terminate instance.
mybiodb$terminate()
```

MassSqliteConn-class *Class for handling a Mass spectrometry database in SQLite format.*

Description

This is the connector class for a MASS SQLite database.

Methods

`getChromCol(ids = NULL)` :

Gets a list of chromatographic columns contained in this database.

ids: A character vector of entry identifiers (i.e.: accession numbers). Used to restrict the set of entries on which to run the algorithm.

Returned value : A data.frame with two columns, one for the ID 'id' and another one for the title 'title'.

See Also

Super classes BiodbMassdbConn and SqliteConn.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get path to LCMS database example file
lcmsdb <- system.file("extdata", "massbank_extract.sqlite", package="biodb")

# Create a connector
conn <- mybiodb$getFactory()$createConn('mass.sqlite', url=lcmsdb)

# Get an entry
e <- conn$getEntry('34.pos.col12.0.78')

# Terminate instance.
mybiodb$terminate()
```

MassSqliteEntry-class *Mass spectra SQLite entry class.*

Description

This is the entry class for a Mass spectra SQLite database.

See Also

Super class BiodbListEntry.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get path to LCMS database example file
lcmsdb <- system.file("extdata", "massbank_extract.sqlite", package="biodb")

# Create a connector
conn <- mybiodb$getFactory()$createConn('mass.sqlite', url=lcmsdb)

# Get an entry
e <- conn$getEntry('34.pos.col12.0.78')

# Terminate instance.
mybiodb$terminate()
```

newInst *Create a new BiodbMain instance.*

Description

Instantiates a new BiodbMain object by calling the constructor.

Usage

```
newInst(...)
```

Arguments

...	The parameters to pass to the BiodbMain constructor. See BiodbMain.

Value

A new BiodbMain instance.

See Also

BiodbMain.

Examples

```
# Create a new BiodbMain instance:
mybiodb <- biodb::newInst()

# Terminate the instance:
mybiodb$terminate()
```

Progress *Progress class.*

Description

A class for informing user about the progress of a process.

Details

This class displays progress of a process to user, and sends notifications of this progress to observers too.

Methods

Public methods:

- Progress$new()
- Progress$increment()
- Progress$clone()

Method `new()`**:** Constructor.

Usage:

```
Progress$new(biodb = NULL, msg, total)
```

Arguments:

`biodb` A BiodbMain instance that will be used to notify observers of progress.

`msg` The message to display to the user.

`total` The total number of elements to process.

Returns: A new instance.

Method `increment()`**:** Increment progress.

Usage:

```
Progress$increment()
```

Returns: Nothing.

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

Usage:

```
Progress$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create an instance
prg <- biodb::Progress$new(msg='Processing data.', total=10)

# Processing
for (i in seq_len(10)) {
    print("Doing something.")
    prg$increment()
}
```

Range *Range class.*

Description

Range class.

Range class.

Details

A class for storing min/max range or value/tolerance.

Methods

Public methods:

- `Range$new()`
- `Range$getValue()`
- `Range$getMin()`
- `Range$getMax()`
- `Range$getMinMax()`
- `Range$getDelta()`
- `Range$getPpm()`
- `Range$clone()`

Method `new()`**:** Constructor.

Usage:

```
Range$new(
  min = NULL,
  max = NULL,
  value = NULL,
  delta = NULL,
  ppm = NULL,
  tol = NULL,
  tolType = c("delta", "plain", "ppm")
)
```

Arguments:

`min` The minimum value of the range.

`max` The maximum value of the range.

`value` The value.

`delta` The delta tolerance.

`ppm` The PPM tolerance.

`tol` The tolerance value, whose type (ppm or delta) is specified by the "tolType" parameter.

`tolType` The type of the tolerance value specified by the "tol" parameter.

Returns: A new instance.

Examples:

```
# Create an instance from min and max:
Range$new(min=1.2, max=1.5)
```

Method `getValue()`**:** Gets the middle value of the range.

Usage:

```
Range$getValue()
```

Returns: The middle value.

Method `getMin()`**:** Gets the minimum value of the range.

Usage:

```
Range$getMin()
```

Returns: The minimum value.

Method `getMax()`**:** Gets the maximum value of the range.

Usage:

```
Range$getMax()
```

Returns: The maximum value.

Method `getMinMax()`**:** Get the min/max range.

Usage:

```
Range$getMinMax()
```

Returns: A list containing two fields: "min" and "max.

Method `getDelta()`**:** Gets the delta tolerance of the range.

Usage:

```
Range$getDelta()
```

Returns: The delta tolerance.

Method `getPpm()`**:** Gets the PPM tolerance of the range.

Usage:

```
Range$getPpm()
```

Returns: The tolerance in PPM.

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

Usage:

```
Range$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Convert a min/max range into a value/ppm tolerance:
rng <- Range$new(min=0.4, max=0.401)
value <- rng$getValue()
ppm <- rng$getPpm()


## ------------------------------------------------
## Method `Range$new`
## ------------------------------------------------

# Create an instance from min and max:
Range$new(min=1.2, max=1.5)
```

`runGenericTests` *Run generic tests.*

Description

This function must be used in tests on all connector classes, before any specific tests.

Usage

```
runGenericTests(
  conn,
  opt = NULL,
  short = TRUE,
  long = FALSE,
  maxShortTestRefEntries = 1
)
```

Arguments

`conn`	A valid biodb connector.
`opt`	A set of options to pass to the test functions.
`short`	Run short tests.
`long`	Run long tests.
`maxShortTestRefEntries`	The maximum number of reference entries to use in short tests.

Value

Nothing.

Examples

```
# Instantiate a Biodb instance for testing
biodb <- biodb::createBiodbTestInstance()

# Create a connector instance
lcmsdb <- system.file("extdata", "massbank_extract.tsv", package="biodb")
conn <- biodb$getFactory()$createConn('mass.csv.file', lcmsdb)

# Run generic tests

biodb::runGenericTests(conn)


# Terminate the instance
biodb$terminate()
```

SqliteConn-class *SQLite connector class.*

Description

This is the abstract connector class for all SQLite databases.

Methods

`getEntryContentFromDb(entry.id)` :

Get the contents of entries directly from the database. A direct request or an access to the database will be made in order to retrieve the contents. No access to the biodb cache system will be made.

entry.id: A character vector with the IDs of entries to retrieve.

Returned value: A character vector, the same size of entry.id, with contents of the requested entries. An NA value will be set for the content of each entry for which the retrieval failed.

`isSearchableByField(field)` :

Tests if a field can be used to search entries when using methods searchByName() and searchCompound().

field: The name of the field.

Returned value: Returns TRUE if the database is searchable using the specified field, FALSE otherwise.

See Also

Super classes `BiodbConn`, `BiodbWritable`, `BiodbEditable`, and sub-classes `CompSqliteConn`, and `MassSqliteConn`.

Examples

```
# Create an instance with default settings:
mybiodb <- biodb::newInst()

# Get a connector that inherits from SqliteConn:
chebi_file <- system.file("extdata", "chebi_extract.sqlite", package="biodb")
conn <- mybiodb$getFactory()$createConn('comp.sqlite', url=chebi_file)

# Get an entry
e <- conn$getEntry('1018')

# Terminate instance.
mybiodb$terminate()
```

`testContext` *Set a test context.*

Description

Define a context for tests using testthat framework. In addition to calling `testthat::context()`.

Usage

```
testContext(text)
```

Arguments

`text`	The text to print as test context.

Value

No value returned.

Examples

```
# Define a context before running tests:
biodb::testContext("Test my database connector.")

# Instantiate a BiodbMain instance for testing
biodb <- biodb::createBiodbTestInstance()

# Terminate the instance
biodb$terminate()
```

testThat *Run a test.*

Description

Run a test function, using testthat framework. In addition to calling `testthat::test_that()`.

Usage

```
testThat(msg, fct, biodb = NULL, conn = NULL, opt = NULL)
```

Arguments

msg	The test message.
fct	The function to test.
biodb	A valid BiodbMain instance to be passed to the test function.
conn	A connector instance to be passed to the test function.
opt	A set of options to pass to the test function.

Value

No value returned.

Examples

```
# Define a context before running tests:
biodb::testContext("Test my database connector.")

# Instantiate a BiodbMain instance for testing
biodb <- biodb::createBiodbTestInstance()

# Define a test function
my_test_function <- function(biodb) {
  # Do my tests...
}

# Run test
biodb::testThat("My test works", my_test_function, biodb=biodb)

# Terminate the instance
biodb$terminate()
```

upgradeExtPkg *Upgrading an existing extension package for biodb.*

Description

Upgrades some of the files previously generated (.gitignore, .travis.yml, .Rbuildignore, Makefile, etc) to the latest versions.

Usage

```
upgradeExtPkg(...)
```

Arguments

...	Parameters passed to ExtPackage constructor.

Value

Nothing.

Examples

```
# Generate a new package:
pkgFolder <- file.path(tempfile(), 'biodbFoo')
dir.create(pkgFolder, recursive=TRUE)
biodb::genNewExtPkg(path=pkgFolder, dbName='foo.db',
                    dbTitle='Foo database', rcpp=TRUE,
                    connType='mass', entryType='txt', downloadable=TRUE,
                    remote=TRUE)

# Upgrade it later
biodb::upgradeExtPkg(path=pkgFolder)
```

warn *Throw a warning and log it too.*

Description

Throws a warning and logs it too with biodb logger.

Usage

```
warn(...)
```

Arguments

...	Values to be passed to sprintf().

Value

Nothing.

Examples

```
# Throws a warning:
tryCatch(biodb::warn('Index is %d.', 10), warning=function(w){w$message})
```

`warn0` *Throw a warning and log it too.*

Description

Throws a warning and logs it too with biodb logger, using paste0().

Usage

```
warn0(...)
```

Arguments

`...`	Values to be passed to paste0().

Value

Nothing.

Examples

```
# Throws a warning:
tryCatch(biodb::warn0('Index is ', 10, '.'), warning=function(w){w$message})
```

Index