Package 'biodb'

October 1, 2026

Title Biodb, a Library and a Development Framework for Connecting to Chemical and Biological Databases

Version 1.21.0

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.

URL https://gitlab.com/rbiodb/biodb

BugReports https://gitlab.com/rbiodb/biodb/-/issues

biocViews Software, Infrastructure, DataImport, KEGG

Depends R (>= 4.1.0)

License AGPL-3

Encoding UTF-8

VignetteBuilder knitr

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

Imports R6, RSQLite, Rcpp, XML, chk, fscache (>= 1.0.2), jsonlite, lgr, lifecycle, methods, openssl, plyr, progress, rappdirs, sched (>= 1.0.1), sqlq, stats, stringr, tools, withr, yaml

LinkingTo Rcpp, testthat

NeedsCompilation yes

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Collate 'BiodbConfig.R' 'BiodbConnBase.R' 'BiodbConn.R' 'BiodbEntry.R' 'BiodbCsvEntry.R' 'BiodbDbInfo.R' 'BiodbDbsInfo.R' 'BiodbEntryField.R' 'BiodbMain.R' 'BiodbEntryFields.R' 'BiodbFactory.R' 'BiodbXmlEntry.R' 'BiodbHtmlEntry.R' 'BiodbJsonEntry.R' 'BiodbListEntry.R' 'BiodbTxtEntry.R' 'BiodbSdfEntry.R' 'BiodbTestMsgAck.R' 'CsvFileConn.R' 'CompCsvFileConn.R' 'CompCsvFileEntry.R' 'SqliteConn.R' 'CompSqliteConn.R' 'CompSqliteEntry.R' 'FileTemplate.R'

'MassCsvFileConn.R' 'MassCsvFileEntry.R' 'MassSqliteConn.R'
'MassSqliteEntry.R' 'Progress.R' 'Range.R' 'RcppExports.R'
'TestRefEntries.R' 'catch-routine-registration.R'
'fcts_biodb.R' 'fcts_deprecated.R' 'fcts_mass.R' 'fcts_misc.R'
'generic_tests.R' 'package.R' 'spec-dist.R' 'test_framework.R'

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

git_branch devel

git_last_commit cd5fba6

git_last_commit_date 2026-04-28

Repository Bioconductor 3.24

Date/Publication 2026-10-01

Author Pierrick Roger [aut, cre] (ORCID:
<https://orcid.org/0000-0001-8177-4873>),
Alexis Delabrière [ctb] (ORCID:
<https://orcid.org/0000-0003-3308-4549>)

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

Contents

biodb-package . . . 3
abstractClass . . . 4
abstractMethod . . . 4
BiodbConfig . . . 5
BiodbConn . . . 9
BiodbConnBase . . . 28
BiodbCsvEntry . . . 34
BiodbDbInfo . . . 35
BiodbDbsInfo . . . 36
BiodbEntry . . . 38
BiodbEntryField . . . 46
BiodbEntryFields . . . 53
BiodbFactory . . . 57
BiodbHtmlEntry . . . 61
BiodbJsonEntry . . . 62
BiodbListEntry . . . 63
BiodbMain . . . 63
BiodbSdfEntry . . . 71
BiodbTestMsgAck . . . 72
BiodbTxtEntry . . . 73
BiodbXmlEntry . . . 74
checkDeprecatedCacheFolders . . . 75
closeMatchPpm . . . 75
CompCsvFileConn . . . 76
CompCsvFileEntry . . . 77
CompSqliteConn . . . 78
CompSqliteEntry . . . 79
connNameToClassPrefix . . . 80
createBiodbTestInstance . . . 80
CsvFileConn . . . 81

df2str . . . 84
error . . . 85
error0 . . . 85
FileTemplate . . . 86
getConnClassName . . . 87
getConnTypes . . . 88
getDefaultCacheDir . . . 88
getEntryClassName . . . 89
getEntryTypes . . . 89
getLogger . . . 90
listTestRefEntries . . . 90
loadFileContents . . . 91
logDebug . . . 91
logDebug0 . . . 92
logInfo . . . 92
logInfo0 . . . 93
logTrace . . . 93
logTrace0 . . . 94
lst2str . . . 94
MassCsvFileConn . . . 95
MassCsvFileEntry . . . 97
MassSqliteConn . . . 98
MassSqliteEntry . . . 99
newInst . . . 100
Progress . . . 100
Range . . . 101
runGenericTests . . . 104
SqliteConn . . . 105
testContext . . . 106
TestRefEntries . . . 107
testThat . . . 109
warn . . . 110
warn0 . . . 110

Index **111**

`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> (ORCID)

Other contributors:

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

See Also

BiodbMain, BiodbConfig, BiodbFactory, BiodbDbsInfo, BiodbEntryFields.

---

abstractClass *Declares a class as abstract.*

Description

Forbids instantiation of an abstract class. This method must be called from within a constructor of an abstract class. It will throw an error if a direct call is made to this constructor.

Usage

```
abstractClass(cls, obj)
```

Arguments

cls	The name of the abstract class to check.
obj	The object being instantiated.

Value

Nothing.

---

abstractMethod *Declares a method as abstract*

Description

This method must be called from within the abstract method.

Usage

```
abstractMethod(obj)
```

Arguments

obj	The object on which the abstract method is called.

Value

Nothing.

BiodbConfig *A class for storing configuration values.*

Description

A class for storing configuration values.

A class for storing configuration values.

Details

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

Public methods:

- `BiodbConfig$new()`
- `BiodbConfig$getKeys()`
- `BiodbConfig$getTitle()`
- `BiodbConfig$getDescription()`
- `BiodbConfig$getDefaultValue()`
- `BiodbConfig$hasKey()`
- `BiodbConfig$isDefined()`
- `BiodbConfig$isEnabled()`
- `BiodbConfig$get()`
- `BiodbConfig$set()`
- `BiodbConfig$reset()`
- `BiodbConfig$enable()`
- `BiodbConfig$disable()`
- `BiodbConfig$print()`
- `BiodbConfig$listKeys()`
- `BiodbConfig$getAssocEnvVar()`
- `BiodbConfig$define()`
- `BiodbConfig$notifyNewObservers()`
- `BiodbConfig$terminate()`
- `BiodbConfig$clone()`

Method `new()`**:** New instance initializer. No BiodbConfig object must not be created directly. Instead, access the config instance through the BiodbMain instance using the getConfig() method.

Usage:

```
BiodbConfig$new(parent)
```

Arguments:

`parent` The BiodbMain instance.

Returns: Nothing.

Method `getKeys()`**:** Get the list of available keys.

Usage:

```
BiodbConfig$getKeys(deprecated = FALSE)
```

Arguments:

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

Returns: A character vector containing the config key names.

Method `getTitle()`**:** Get the title of a key.

Usage:

```
BiodbConfig$getTitle(key)
```

Arguments:

`key` The name of a configuration key.

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

Method `getDescription()`**:** Get the description of a key.

Usage:

```
BiodbConfig$getDescription(key)
```

Arguments:

`key` The name of a configuration key.

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

Method `getDefaultValue()`**:** Get the default value of a key.

Usage:

```
BiodbConfig$getDefaultValue(key, as.chr = FALSE)
```

Arguments:

`key` The name of a configuration key.

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

Returns: The default value for that key.

Method `hasKey()`**:** Test if a key exists.

Usage:

```
BiodbConfig$hasKey(key)
```

Arguments:

`key` The name of a configuration key.

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

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

Usage:

```
BiodbConfig$isDefined(key, fail = TRUE)
```

Arguments:

`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.

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

Method `isEnabled()`**:** Test if a boolean key is set to TRUE. This method will raise an error if the key is not a boolean key.

Usage:

```
BiodbConfig$isEnabled(key)
```

Arguments:

`key` The name of a configuration key.

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

Method `get()`**:** Get the value of a key.

Usage:

```
BiodbConfig$get(key)
```

Arguments:

`key` The name of a configuration key.

Returns: The value associated with the key.

Method `set()`**:** Set the value of a key.

Usage:

```
BiodbConfig$set(key, value)
```

Arguments:

`key` The name of a configuration key.
`value` A value to associate with the key.

Returns: Nothing.

Method `reset()`**:** Reset the value of a key.

Usage:

```
BiodbConfig$reset(key = NULL)
```

Arguments:

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

Returns: Nothing.

Method `enable()`**:** Set a boolean key to TRUE.

Usage:

```
BiodbConfig$enable(key)
```

Arguments:

`key` The name of a configuration key.

Returns: Nothing.

Method `disable()`**:** Set a boolean key to FALSE.

Usage:

```
BiodbConfig$disable(key)
```

Arguments:

`key` The name of a configuration key.

Returns: Nothing.

Method `print()`**:** Print list of configuration keys and their values.

Usage:

```
BiodbConfig$print()
```

Returns: Nothing.

Method `listKeys()`**:** Get the full list of keys as a data frame.

Usage:

```
BiodbConfig$listKeys()
```

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

Method `getAssocEnvVar()`**:** Returns the environment variable associated with this configuration key.

Usage:

```
BiodbConfig$getAssocEnvVar(key)
```

Arguments:

`key` The name of a configuration key.

Returns: The environment variable's value.

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

Usage:

```
BiodbConfig$define(def)
```

Arguments:

`def` The list of key definitions.

Returns: Nothing.

Method `notifyNewObservers()`**:** Called by BiodbMain when a new observer is registered.

Usage:

```
BiodbConfig$notifyNewObservers(obs)
```

Arguments:

`obs` The new observers registered by the BiodbMain instance.

Returns: Nothing.

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

Usage:

```
BiodbConfig$terminate()
```

Arguments:

`BiodbMain` :terminate().

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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 *The mother abstract class of all database connectors.*

Description

The mother abstract class of all database connectors.

The mother abstract class of all database connectors.

Details

This is the super class of all connector classes. All methods defined here are thus common to all connector classes. All connector classes inherit from this abstract 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.*".

The constructor has the following arguments:

id: The identifier of the connector.

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

Super class

biodb::BiodbConnBase -> BiodbConn

Methods

Public methods:

- BiodbConn$new()
- BiodbConn$getBiodb()
- BiodbConn$getId()
- BiodbConn$print()
- BiodbConn$correctIds()
- BiodbConn$getEntry()
- BiodbConn$getCacheFile()
- BiodbConn$getEntryContent()
- BiodbConn$getEntryContentFromDb()
- BiodbConn$getEntryContentRequest()
- BiodbConn$getEntryIds()
- BiodbConn$getNbEntries()
- BiodbConn$isEditable()
- BiodbConn$editingIsAllowed()
- BiodbConn$allowEditing()
- BiodbConn$disallowEditing()
- BiodbConn$setEditingAllowed()
- BiodbConn$addNewEntry()
- BiodbConn$isWritable()
- BiodbConn$allowWriting()
- BiodbConn$disallowWriting()
- BiodbConn$setWritingAllowed()
- BiodbConn$writingIsAllowed()
- BiodbConn$write()
- BiodbConn$isSearchableByField()
- BiodbConn$getSearchableFields()
- BiodbConn$searchForEntries()
- BiodbConn$searchByName()
- BiodbConn$isDownloadable()
- BiodbConn$isDownloaded()
- BiodbConn$requiresDownload()
- BiodbConn$getDownloadPath()
- BiodbConn$setDownloadedFile()
- BiodbConn$isExtracted()
- BiodbConn$download()
- BiodbConn$isRemotedb()
- BiodbConn$isCompounddb()
- BiodbConn$searchCompound()
- BiodbConn$annotateMzValues()
- BiodbConn$isMassdb()

- BiodbConn$checkDb()
- BiodbConn$getAllVolatileCacheEntries()
- BiodbConn$getAllCacheEntries()
- BiodbConn$deleteAllEntriesFromVolatileCache()
- BiodbConn$deleteAllEntriesFromPersistentCache()
- BiodbConn$deleteWholePersistentCache()
- BiodbConn$deleteAllCacheEntries()
- BiodbConn$getCacheId()
- BiodbConn$makesRefToEntry()
- BiodbConn$makeRequest()
- BiodbConn$getEntryImageUrl()
- BiodbConn$getEntryPageUrl()
- BiodbConn$getChromCol()
- BiodbConn$getMatchingMzField()
- BiodbConn$setMatchingMzField()
- BiodbConn$getMzValues()
- BiodbConn$getNbPeaks()
- BiodbConn$filterEntriesOnRt()
- BiodbConn$searchForMassSpectra()
- BiodbConn$searchMsEntries()
- BiodbConn$searchMsPeaks()
- BiodbConn$msmsSearch()
- BiodbConn$collapseResultsDataFrame()
- BiodbConn$searchMzRange()
- BiodbConn$searchMzTol()
- BiodbConn$clone()

Method `new()`**:** New instance initializer. Connector objects must not be created directly. Instead, you create new connector instances through the BiodbFactory instance.

Usage:

```
BiodbConn$new(id = NA_character_, cache.id = NA_character_, bdb, ...)
```

Arguments:

`id` The ID of the connector instance.

`cache.id` The Cache ID of the connector instance.

`bdb` The BiodbMain instance.

`...` Remaining arguments will be passed to the constructor of the super class.

Returns: Nothing.

Method `getBiodb()`**:** Returns the biodb main class instance to which this object is attached.

Usage:

```
BiodbConn$getBiodb()
```

Returns: The main biodb instance.

Method `getId()`**:** Get the identifier of this connector.

Usage:

```
BiodbConn$getId()
```

Returns: The identifier of this connector.

Method `print()`**:** Prints a description of this connector.

Usage:

```
BiodbConn$print()
```

Returns: Nothing.

Method `correctIds()`**:** Correct a vector of IDs by formatting them to the database official format, if required and possible.

Usage:

```
BiodbConn$correctIds(ids)
```

Arguments:

`ids` A character vector of IDs.

Returns: The vector of IDs corrected.

Method `getEntry()`**:** Return the entry corresponding to this ID. You can pass a vector of IDs, and you will get a list of entries.

Usage:

```
BiodbConn$getEntry(id, drop = TRUE, nulls = TRUE)
```

Arguments:

`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.

Returns: 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.

Method `getCacheFile()`**:** Get the path to the persistent cache file.

Usage:

```
BiodbConn$getCacheFile(entry.id)
```

Arguments:

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

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

Method `getEntryContent()`**:** Get the contents of database entries from IDs (accession numbers).

Usage:

```
BiodbConn$getEntryContent(id)
```

Arguments:

`id` A character vector of entry IDs.

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

Method `getEntryContentFromDb()`**:** 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.

Usage:

```
BiodbConn$getEntryContentFromDb(entry.id)
```

Arguments:

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

Returns: 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.

Method `getEntryContentRequest()`**:** Gets the URL to use in order to get the contents of the specified entries.

Usage:

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

Arguments:

`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.

Returns: A vector of URL strings.

Method `getEntryIds()`**:** Get entry identifiers from the database. More arguments can be given, depending on implementation in specific databases. For mass databases the ms.level argument can also be set.

Usage:

```
BiodbConn$getEntryIds(max.results = 0, ...)
```

Arguments:

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

`...` Arguments specific to connectors.

Returns: 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.

Method `getNbEntries()`**:** Get the number of entries contained in this database.

Usage:

```
BiodbConn$getNbEntries(count = FALSE)
```

Arguments:

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

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

Method `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.

Usage:

```
BiodbConn$isEditable()
```

Returns: Returns TRUE if the database is editable.

Method `editingIsAllowed()`**:** Tests if editing is allowed.

Usage:

```
BiodbConn$editingIsAllowed()
```

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

Method `allowEditing()`**:** Allows editing for this database.

Usage:

```
BiodbConn$allowEditing()
```

Returns: Nothing.

Method `disallowEditing()`**:** Disallows editing for this database.

Usage:

```
BiodbConn$disallowEditing()
```

Returns: Nothing.

Method `setEditingAllowed()`**:** Allow or disallow editing for this database.

Usage:

```
BiodbConn$setEditingAllowed(allow)
```

Arguments:

`allow` A logical value.

Returns: Nothing.

Method `addNewEntry()`**:** 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().

Usage:

```
BiodbConn$addNewEntry(entry)
```

Arguments:

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

Returns: Nothing.

Method `isWritable()`**:** Tests if this connector is able to write into the database. If this connector is writable, then you can call allowWriting() to enable writing.

Usage:

```
BiodbConn$isWritable()
```

Returns: Returns TRUE if the database is writable.

Method `allowWriting()`**:** Allows the connector to write into this database.

Usage:

```
BiodbConn$allowWriting()
```

Returns: Nothing.

Method `disallowWriting()`**:** Disallows the connector to write into this database.

Usage:

```
BiodbConn$disallowWriting()
```

Returns: Nothing.

Method `setWritingAllowed()`**:** Allows or disallows writing for this database.

Usage:

```
BiodbConn$setWritingAllowed(allow)
```

Arguments:

`allow` If set to TRUE, allows writing.

Returns: Nothing.

Method `writingIsAllowed()`**:** Tests if the connector has access right to the database.

Usage:

```
BiodbConn$writingIsAllowed()
```

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

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

Usage:

```
BiodbConn$write()
```

Returns: Nothing.

Method `isSearchableByField()`**:** Tests if a field can be used to search entries when using method searchForEntries().

Usage:

```
BiodbConn$isSearchableByField(field = NULL, field.type = NULL)
```

Arguments:

`field` The name of the field.

`field.type` The field type.

Returns: Returns TRUE if the database is searchable using the specified field or searchable by any field of the specified type, FALSE otherwise.

Method `getSearchableFields()`**:** Get the list of all searchable fields.

Usage:

```
BiodbConn$getSearchableFields()
```

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

Method `searchForEntries()`**:** Searches the database for entries whose name matches the specified name. Returns a character vector of entry IDs.

Usage:

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

Arguments:

`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.

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

Method `searchByName()`**:** DEPRECATED. Use searchForEntries() instead.

Usage:

```
BiodbConn$searchByName(name, max.results = 0)
```

Arguments:

`name` A character value to search inside name fields.

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

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

Method `isDownloadable()`**:** Tests if the connector can download the database.

Usage:

```
BiodbConn$isDownloadable()
```

Returns: Returns TRUE if the database is downloadable.

Method `isDownloaded()`**:** Tests if the database has been downloaded.

Usage:

```
BiodbConn$isDownloaded()
```

Returns: TRUE if the database content has already been downloaded.

Method `requiresDownload()`**:** Tests if the connector requires the download of the database.

Usage:

```
BiodbConn$requiresDownload()
```

Returns: TRUE if the connector requires download of the database.

Method `getDownloadPath()`**:** Gets the path where the downloaded content is written.

Usage:

```
BiodbConn$getDownloadPath()
```

Returns: The path where the downloaded database is written.

Method `setDownloadedFile()`**:** Set the downloaded file into the cache.

Usage:

```
BiodbConn$setDownloadedFile(src, action = c("copy", "move"))
```

Arguments:

`src` Path to the downloaded file.

`action` Specifies if files have to be moved or copied into the cache.

Returns: Nothing.

Method `isExtracted()`**:** Tests if the downloaded database has been extracted (in case the database needs extraction).

Usage:

```
BiodbConn$isExtracted()
```

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

Method `download()`**:** Downloads the database content locally.

Usage:

```
BiodbConn$download()
```

Returns: Nothing.

Method `isRemotedb()`**:** Tests if the connector is connected to a remote database.

Usage:

```
BiodbConn$isRemotedb()
```

Returns: Returns TRUE if the database is a remote database."

Method `isCompounddb()`**:** Tests if the connector's database is a compound database.

Usage:

```
BiodbConn$isCompounddb()
```

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

Method `searchCompound()`**:** 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.

Usage:

```
BiodbConn$searchCompound(
  name = NULL,
  mass = NULL,
  mass.field = NULL,
  mass.tol = 0.01,
  mass.tol.unit = "plain",
  max.results = 0
)
```

Arguments:

`name` The name of a compound to search for.

`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.

`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.

Returns: A character vector of entry IDs."

Method `annotateMzValues()`**:** 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.

Usage:

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

```
  max.results = 3,
  mz.col = "mz",
  fields = NULL,
  prefix = NULL,
  insert.input.values = TRUE,
  fieldsLimit = 0
)
```

Arguments:

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

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

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

`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'.

`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.

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

`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.

`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.

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

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

Returns: 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.

Method `isMassdb()`**:** Tests if the connector's database is a mass spectra database.

Usage:

```
BiodbConn$isMassdb()
```

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

Method `checkDb()`**:** Checks that the database is correct by trying to retrieve all its entries.

Usage:

```
BiodbConn$checkDb()
```

Returns: Nothing.

Method `getAllVolatileCacheEntries()`**:** Get all entries stored in the memory cache (volatile cache).

Usage:

```
BiodbConn$getAllVolatileCacheEntries()
```

Returns: A list of BiodbEntry instances.

Method `getAllCacheEntries()`**:** This method is deprecated. Use getAllVolatileCacheEntries() instead.

Usage:

```
BiodbConn$getAllCacheEntries()
```

Returns: All entries cached in memory.

Method `deleteAllEntriesFromVolatileCache()`**:** Delete all entries from the volatile cache (memory cache).

Usage:

```
BiodbConn$deleteAllEntriesFromVolatileCache()
```

Returns: Nothing.

Method `deleteAllEntriesFromPersistentCache()`**:** Delete all entries from the persistent cache (disk cache).

Usage:

```
BiodbConn$deleteAllEntriesFromPersistentCache(deleteVolatile = TRUE)
```

Arguments:

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

Returns: Nothing.

Method `deleteWholePersistentCache()`**:** Delete all files associated with this connector from the persistent cache (disk cache).

Usage:

```
BiodbConn$deleteWholePersistentCache(deleteVolatile = TRUE)
```

Arguments:

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

Returns: Nothing.

Method `deleteAllCacheEntries()`**:** Delete all entries from the memory cache. This method is deprecated, please use deleteAllEntriesFromVolatileCache() instead.

Usage:

```
BiodbConn$deleteAllCacheEntries()
```

Returns: Nothing.

Method `getCacheId()`**:** Gets the ID used by this connector in the disk cache.

Usage:

```
BiodbConn$getCacheId()
```

Returns: The cache ID of this connector.

Method `makesRefToEntry()`**:** Tests if some entry of this database makes reference to another entry of another database.

Usage:

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

Arguments:

`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`.

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

Method `makeRequest()`**:** Makes a sched::Request instance using the passed parameters.

Usage:

```
BiodbConn$makeRequest(...)
```

Arguments:

`...` Those parameters are passed to the initializer of sched::Request.

Returns: The sched::Request instance.

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

Usage:

```
BiodbConn$getEntryImageUrl(entry.id)
```

Arguments:

`entry.id` A character vector containing entry IDs.

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

Method `getEntryPageUrl()`**:** Gets the URL to the page of the entry on the database web site.

Usage:

```
BiodbConn$getEntryPageUrl(entry.id)
```

Arguments:

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

Returns: A list of sched::URL objects, the same length as `entry.id`.

Method `getChromCol()`**:** Gets a list of chromatographic columns contained in this database.

Usage:

```
BiodbConn$getChromCol(ids = NULL)
```

Arguments:

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

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

Method `getMatchingMzField()`**:** Gets the field to use for M/Z matching.

Usage:

```
BiodbConn$getMatchingMzField()
```

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

Method `setMatchingMzField()`**:** Sets the field to use for M/Z matching.

Usage:

```
BiodbConn$setMatchingMzField(field = c("peak.mztheo", "peak.mzexp"))
```

Arguments:

`field` The field to use for matching.

Returns: Nothing.

Method `getMzValues()`**:** Gets a list of M/Z values contained inside the database.

Usage:

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

Arguments:

`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.

Returns: A numeric vector containing M/Z values.

Method `getNbPeaks()`**:** Gets the number of peaks contained in the database.

Usage:

```
BiodbConn$getNbPeaks(mode = NULL, ids = NULL)
```

Arguments:

`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.

Returns: The number of peaks, as an integer.

Method `filterEntriesOnRt()`**:** Filters a list of entries on retention time values.

Usage:

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

Arguments:

`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.

Returns: A character vector containing entry IDs after filtering.

Method `searchForMassSpectra()`**:** 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.

Usage:

```
BiodbConn$searchForMassSpectra(
  mz.min = NULL,
  mz.max = NULL,
  mz = NULL,
  mz.tol = NULL,
  mz.tol.unit = c("plain", "ppm"),
  rt = NULL,
  rt.unit = c("s", "min"),
  rt.tol = NULL,
  rt.tol.exp = NULL,
  chrom.col.ids = NULL,
  precursor = FALSE,
  min.rel.int = 0,
  ms.mode = NULL,
  max.results = 0,
  ms.level = 0,
  include.ids = NULL
)
```

Arguments:

`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`.

`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'.

`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'.

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

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

`include.ids` A list of IDs to which to restrict the final results. All IDs that are not in this list will be excluded.

Returns: A character vector of spectra IDs.

Method `searchMsEntries()`**:** DEPRECATED. Use searchForMassSpectra() instead.

Usage:

```
BiodbConn$searchMsEntries(
  mz.min = NULL,
  mz.max = NULL,
  mz = NULL,
  mz.tol = NULL,
  mz.tol.unit = c("plain", "ppm"),
  rt = NULL,
  rt.unit = c("s", "min"),
  rt.tol = NULL,
  rt.tol.exp = NULL,
  chrom.col.ids = NULL,
  precursor = FALSE,
  min.rel.int = 0,
  ms.mode = NULL,
  max.results = 0,
  ms.level = 0
)
```

Arguments:

`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`.

`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'.

`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'.

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

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

Returns: A character vector of spectra IDs.

Method `searchMsPeaks()`**:** For each M/Z value, searches for matching MS spectra and returns the matching peaks.

Usage:

```
BiodbConn$searchMsPeaks(
  input.df = NULL,
  mz = NULL,
  mz.tol = NULL,
  mz.tol.unit = c("plain", "ppm"),
  min.rel.int = 0,
  ms.mode = NULL,
  ms.level = 0,
  max.results = 0,
  chrom.col.ids = NULL,
  rt = NULL,
  rt.unit = c("s", "min"),
  rt.tol = NULL,
  rt.tol.exp = NULL,
  precursor = FALSE,
  precursor.rt.tol = NULL,
  insert.input.values = TRUE,
  prefix = NULL,
  compute = TRUE,
  fields = NULL,
  fieldsLimit = 0,
  input.df.colnames = c(mz = "mz", rt = "rt"),
  match.rt = FALSE
)
```

Arguments:

`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.

Returns: 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`.

Method `msmsSearch()`**:** Searches MSMS spectra matching a template spectrum. The mz.tol parameter is applied on the precursor search.

Usage:

```
BiodbConn$msmsSearch(
  spectrum,
  precursor.mz,
  mz.tol,
  mz.tol.unit = c("plain", "ppm"),
  ms.mode,
  npmin = 2,
  dist.fun = c("wcosine", "cosine", "pkernel", "pbachtttarya"),
  msms.mz.tol = 3,
  msms.mz.tol.min = 0.005,
  max.results = 0
)
```

Arguments:

`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.

Returns: 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.

Method `collapseResultsDataFrame()`**:** Collapse rows of a results data frame, by outputing a data frame with only one row for each MZ/RT value.

Usage:

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

Arguments:

`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.

Returns: A data frame with rows collapsed."

Method `searchMzRange()`**:** Find spectra in the given M/Z range. Returns a list of spectra IDs.

Usage:

```
BiodbConn$searchMzRange(
  mz.min,
  mz.max,
  min.rel.int = 0,
  ms.mode = NULL,
  max.results = 0,
  precursor = FALSE,
  ms.level = 0
)
```

Arguments:

`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`.

`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'.

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

`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.

Returns: A character vector of spectra IDs.

Method `searchMzTol()`**:** Find spectra containg a peak around the given M/Z value. Returns a character vector of spectra IDs.

Usage:

```
BiodbConn$searchMzTol(
  mz,
  mz.tol,
  mz.tol.unit = "plain",
  min.rel.int = 0,
  ms.mode = NULL,
```

```
  max.results = 0,
  precursor = FALSE,
  ms.level = 0
)
```

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'.

`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'.

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

`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.

Returns: A character vector of spectra IDs.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Super class `BiodbConnBase`, and `BiodbFactory` class.

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

Description

Base class of BiodbConn for encapsulating all needed information for database access.

Base class of BiodbConn for encapsulating all needed information for database access.

Details

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.

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

Public methods:

- BiodbConnBase$new()
- BiodbConnBase$print()
- BiodbConnBase$hasProp()
- BiodbConnBase$getPropSlots()
- BiodbConnBase$hasPropSlot()
- BiodbConnBase$propExists()
- BiodbConnBase$isSlotProp()
- BiodbConnBase$getPropValSlot()
- BiodbConnBase$updatePropertiesDefinition()
- BiodbConnBase$getEntryFileExt()
- BiodbConnBase$getDbClass()
- BiodbConnBase$getConnClassName()
- BiodbConnBase$getConnClass()
- BiodbConnBase$getEntryClassName()
- BiodbConnBase$getEntryClass()
- BiodbConnBase$getEntryIdField()
- BiodbConnBase$getPropertyValue()
- BiodbConnBase$setPropertyValue()
- BiodbConnBase$setPropValSlot()
- BiodbConnBase$getBaseUrl()

- BiodbConnBase$setBaseUrl()
- BiodbConnBase$getWsUrl()
- BiodbConnBase$setWsUrl()
- BiodbConnBase$getToken()
- BiodbConnBase$setToken()
- BiodbConnBase$getName()
- BiodbConnBase$getEntryContentType()
- BiodbConnBase$getSchedulerNParam()
- BiodbConnBase$setSchedulerNParam()
- BiodbConnBase$getSchedulerTParam()
- BiodbConnBase$setSchedulerTParam()
- BiodbConnBase$getUrls()
- BiodbConnBase$getUrl()
- BiodbConnBase$setUrl()
- BiodbConnBase$getXmlNs()
- BiodbConnBase$clone()

Method `new()`**:** New instance initializer. Connector objects must not be created directly. Instead, you create new connector instances through the BiodbFactory instance.

Usage:

```
BiodbConnBase$new(other = NULL, db.class = NULL, properties = NULL, cfg = NULL)
```

Arguments:

`other` Another BiodbConnBase instance as a model from which to copy property values.

`db.class` The class of the connector (i.e.: "mass.csv.file").

`properties` Some new values for the properties.

`cfg` The BiodbConfig instance from which will be taken some property values.

Returns: Nothing.

Method `print()`**:** Prints a description of this connector.

Usage:

```
BiodbConnBase$print()
```

Returns: Nothing.

Method `hasProp()`**:** Tests if this connector has a property.

Usage:

```
BiodbConnBase$hasProp(name)
```

Arguments:

`name` The name of the property to check.

Returns: Returns true if the property `name` exists.

Method `getPropSlots()`**:** Gets the slot fields of a property.

Usage:

```
BiodbConnBase$getPropSlots(name)
```

Arguments:

`name` The name of a property.

Returns: Returns a character vector containing all slot names defined.

Method `hasPropSlot()`**:** Tests if a slot property has a specific slot.

Usage:

```
BiodbConnBase$hasPropSlot(name, slot)
```

Arguments:

`name` The name of a property.

`slot` The slot name to check.

Returns: Returns TRUE if the property `name` exists and has the slot `slot` defined, and FALSE otherwise."

Method `propExists()`**:** Checks if property exists.

Usage:

```
BiodbConnBase$propExists(name)
```

Arguments:

`name` The name of a property.

Returns: Returns TRUE if the property `name` exists, and FALSE otherwise.

Method `isSlotProp()`**:** Tests if a property is a slot property.

Usage:

```
BiodbConnBase$isSlotProp(name)
```

Arguments:

`name` The name of a property.

Returns: Returns TRUE if the property is a slot propert, FALSE otherwise.

Method `getPropValSlot()`**:** Retrieve the value of a slot of a property.

Usage:

```
BiodbConnBase$getPropValSlot(name, slot, hook = TRUE)
```

Arguments:

`name` The name of a property.

`slot` The slot name inside the property.

`hook` If set to TRUE, enables the calls to hook methods associated with the property. Otherwise, all calls to hook methods are disabled.

Returns: The value of the slot `slot` of the property `name`.

Method `updatePropertiesDefinition()`**:** Update the definition of properties.

Usage:

```
BiodbConnBase$updatePropertiesDefinition(def)
```

Arguments:

`def` A named list of property definitions. The names of the list must be the property names.

Returns: Nothing.

Method `getEntryFileExt()`**:** Returns the entry file extension used by this connector.

Usage:

```
BiodbConnBase$getEntryFileExt()
```

Returns: A character value containing the file extension.

Method `getDbClass()`**:** Gets the Biodb name of the database associated with this connector.

Usage:

```
BiodbConnBase$getDbClass()
```

Returns: A character value containing the Biodb database name.

Method `getConnClassName()`**:** Gets the name of the associated connector OOP class.

Usage:

```
BiodbConnBase$getConnClassName()
```

Returns: Returns the connector OOP class name.

Method `getConnClass()`**:** Gets the associated connector OOP class.

Usage:

```
BiodbConnBase$getConnClass()
```

Returns: Returns the connector OOP class.

Method `getEntryClassName()`**:** Gets the name of the associated entry class.

Usage:

```
BiodbConnBase$getEntryClassName()
```

Returns: Returns the name of the associated entry class.

Method `getEntryClass()`**:** Gets the associated entry class.

Usage:

```
BiodbConnBase$getEntryClass()
```

Returns: Returns the associated entry class.

Method `getEntryIdField()`**:** Gets the name of the corresponding database ID field in entries.

Usage:

```
BiodbConnBase$getEntryIdField()
```

Returns: Returns the name of the database ID field.

Method `getPropertyValue()`**:** Gets a property value.

Usage:

```
BiodbConnBase$getPropertyValue(name, hook = TRUE)
```

Arguments:

`name` The name of the property.

`hook` If set to TRUE, enables the calls to hook methods associated with the property. Otherwise, all calls to hook methods are disabled.

Returns: The value of the property.

Method `setPropertyValue()`**:** Sets the value of a property.

Usage:

```
BiodbConnBase$setPropertyValue(name, value)
```

Arguments:

`name` The name of the property.

`value` The new value to set the property to.

Returns: Nothing.

Method `setPropValSlot()`**:** Set the value of the slot of a property.

Usage:

```
BiodbConnBase$setPropValSlot(name, slot, value, hook = TRUE)
```

Arguments:

`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.

`hook` If set to TRUE, enables the calls to hook methods associated with the property. Otherwise, all calls to hook methods are disabled.

Returns: Nothing.

Method `getBaseUrl()`**:** Returns the base URL.

Usage:

```
BiodbConnBase$getBaseUrl()
```

Returns: THe baae URL.

Method `setBaseUrl()`**:** Sets the base URL.

Usage:

```
BiodbConnBase$setBaseUrl(url)
```

Arguments:

`url` A URL as a character value.

Returns: Nothing.

Method `getWsUrl()`**:** Returns the web sevices URL.

Usage:

```
BiodbConnBase$getWsUrl()
```

Method `setWsUrl()`**:** Sets the web sevices URL.

Usage:

```
BiodbConnBase$setWsUrl(ws.url)
```

Arguments:

`ws.url` A URL as a character value.

Returns: Nothing.

Method `getToken()`**:** Returns the access token.

Usage:

```
BiodbConnBase$getToken()
```

Method `setToken()`**:** Sets the access token.

Usage:

```
BiodbConnBase$setToken(token)
```

Arguments:

`token` The token to use to access the database, as a character value.

Returns: Nothing.

Method `getName()`**:** Returns the full database name.

Usage:

```
BiodbConnBase$getName()
```

Method `getEntryContentType()`**:** Returns the entry content type.

Usage:

```
BiodbConnBase$getEntryContentType()
```

Method `getSchedulerNParam()`**:** Returns the N parameter for the scheduler.

Usage:

```
BiodbConnBase$getSchedulerNParam()
```

Method `setSchedulerNParam()`**:** Sets the N parameter for the scheduler.

Usage:

```
BiodbConnBase$setSchedulerNParam(n)
```

Arguments:

`n` The N parameter as a whole number.

Returns: Nothing.

Method `getSchedulerTParam()`**:** Returns the T parameter for the scheduler.

Usage:

```
BiodbConnBase$getSchedulerTParam()
```

Method `setSchedulerTParam()`**:** Sets the T parameter for the scheduler.

Usage:

```
BiodbConnBase$setSchedulerTParam(t)
```

Arguments:

`t` The T parameter as a whole number.

Returns: Nothing.

Method `getUrls()`**:** Returns the URLs.

Usage:

```
BiodbConnBase$getUrls()
```

Method `getUrl()`**:** Returns a URL.

Usage:

```
BiodbConnBase$getUrl(name)
```

Arguments:

`name` The name of the URL to retrieve.

Returns: The URL as a character value.

Method `setUrl()`**:** Sets a URL.

Usage:

```
BiodbConnBase$setUrl(name, url)
```

Arguments:

`name` The name of the URL to set.

`url` The URL value.

Returns: Nothing.

Method `getXmlNs()`**:** Returns the XML namespace.

Usage:

```
BiodbConnBase$getXmlNs()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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()
```

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

Description

Entry class for content in CSV format.

Entry class for content in CSV format.

Details

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

Super class

biodb::BiodbEntry -> BiodbCsvEntry

Methods

Public methods:

- BiodbCsvEntry$new()
- BiodbCsvEntry$clone()

Method new(): New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
BiodbCsvEntry$new(sep = ",", na.strings = "NA", quotes = "", ...)
```

Arguments:

sep The separator to use in CSV files.

na.strings The strings to recognize as NA values. This is a character vector.

quotes The characters to recognize as quotes. This is a single character value.

... The remaining arguments will be passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbCsvEntry)
```

BiodbDbInfo *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.

Super class

biodb::BiodbConnBase -> BiodbDbInfo

Methods

Public methods:

- BiodbDbInfo$clone()

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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` *A class for describing the available databases.*

Description

A class for describing the available databases.

A class for describing the available databases.

Details

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

Methods

Public methods:

- BiodbDbsInfo$new()
- BiodbDbsInfo$define()
- BiodbDbsInfo$getIds()
- BiodbDbsInfo$isDefined()
- BiodbDbsInfo$checkIsDefined()
- BiodbDbsInfo$get()
- BiodbDbsInfo$getAll()

- BiodbDbsInfo$print()
- BiodbDbsInfo$clone()

Method `new()`**:** New instance initializer. The class must not be instantiated directly. Instead, access the BiodbDbsInfo instance through the BiodbMain instance using the getDbsInfo() method.

Usage:

```
BiodbDbsInfo$new(cfg)
```

Arguments:

`cfg` The BiodbConfig instance.

Returns: Nothing.

Method `define()`**:** Define databases from a structured object, normally loaded from a YAML file.

Usage:

```
BiodbDbsInfo$define(def, package = "biodb")
```

Arguments:

`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.

Returns: Nothing.

Method `getIds()`**:** Gets the database IDs.

Usage:

```
BiodbDbsInfo$getIds()
```

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

Method `isDefined()`**:** Tests if a database is defined.

Usage:

```
BiodbDbsInfo$isDefined(db.id)
```

Arguments:

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

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

Method `checkIsDefined()`**:** Checks if a database is defined. Throws an error if the specified id does not correspond to a defined database.

Usage:

```
BiodbDbsInfo$checkIsDefined(db.id)
```

Arguments:

`db.id` A character vector of database IDs.

Returns: Nothing.

Method `get()`**:** Gets information on a database.

Usage:

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

Arguments:

`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.

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

Method `getAll()`**:** Gets informations on all databases.

Usage:

```
BiodbDbsInfo$getAll()
```

Returns: A list of all BiodbDbInfo instances."

Method `print()`**:** Prints informations about this instance, listing also all databases defined.

Usage:

```
BiodbDbsInfo$print()
```

Returns: Nothing.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

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()
```

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

Description

The mother abstract class of all database entry classes.

The mother abstract class of all database entry classes.

Details

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

Public methods:

- `BiodbEntry$new()`
- `BiodbEntry$parentIsAConnector()`
- `BiodbEntry$getParent()`
- `BiodbEntry$getBiodb()`
- `BiodbEntry$cloneInstance()`
- `BiodbEntry$getId()`
- `BiodbEntry$isNew()`
- `BiodbEntry$getDbClass()`
- `BiodbEntry$setFieldValue()`
- `BiodbEntry$appendFieldValue()`
- `BiodbEntry$getFieldNames()`
- `BiodbEntry$hasField()`
- `BiodbEntry$removeField()`
- `BiodbEntry$getFieldValue()`
- `BiodbEntry$getFieldsByType()`
- `BiodbEntry$getFieldsAsDataframe()`
- `BiodbEntry$getFieldsAsJson()`
- `BiodbEntry$parseContent()`
- `BiodbEntry$computeFields()`
- `BiodbEntry$print()`
- `BiodbEntry$getName()`
- `BiodbEntry$makesRefToEntry()`
- `BiodbEntry$getField()`
- `BiodbEntry$setField()`
- `BiodbEntry$getFieldClass()`
- `BiodbEntry$getFieldDef()`
- `BiodbEntry$getFieldCardinality()`
- `BiodbEntry$fieldHasBasicClass()`
- `BiodbEntry$clone()`

Method `new()`**:** New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
BiodbEntry$new(parent)
```

Arguments:

parent A valid BiodbConn instance.

Returns: Nothing.

Method `parentIsAConnector()`**:** Tests if the parent of this entry is a connector instance.

Usage:

```
BiodbEntry$parentIsAConnector()
```

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

Method `getParent()`**:** Returns the parent instance (A BiodbConn or BiodbFactory object) to which this object is attached.

Usage:

```
BiodbEntry$getParent()
```

Returns: A BiodbConn instance or a BiodbFactory object.

Method `getBiodb()`**:** Returns the biodb main class instance to which this object is attached.

Usage:

```
BiodbEntry$getBiodb()
```

Returns: The main biodb instance.

Method `cloneInstance()`**:** Clones this entry.

Usage:

```
BiodbEntry$cloneInstance(db.class = NULL)
```

Arguments:

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.

Returns: The clone, as a new BiodbEntry instance.

Method `getId()`**:** Gets the entry ID.

Usage:

```
BiodbEntry$getId()
```

Returns: the entry ID, which is the value if the `accession` field.

Method `isNew()`**:** Tests if this entry is new.

Usage:

```
BiodbEntry$isNew()
```

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

Method `getDbClass()`**:** Gets the ID of the database associated with this entry.

Usage:

```
BiodbEntry$getDbClass()
```

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

Method `setFieldValue()`**:** 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.

Usage:

```
BiodbEntry$setFieldValue(field, value)
```

Arguments:

`field` The name of a field.

`value` The value to set.

Returns: Nothing.

Method `appendFieldValue()`**:** 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.

Usage:

```
BiodbEntry$appendFieldValue(field, value)
```

Arguments:

`field` The name of a field.

`value` The value to append.

Returns: Nothing.

Method `getFieldNames()`**:** Gets a list of all fields defined for this entry.

Usage:

```
BiodbEntry$getFieldNames()
```

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

Method `hasField()`**:** Tests if a field is defined in this entry.

Usage:

```
BiodbEntry$hasField(field)
```

Arguments:

`field` The name of a field.

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

Method `removeField()`**:** Removes the specified field from this entry.

Usage:

```
BiodbEntry$removeField(field)
```

Arguments:

`field` The name of a field.

Returns: Nothing.

Method `getFieldValue()`**:** Gets the value of the specified field.

Usage:

```
BiodbEntry$getFieldValue(
  field,
  compute = TRUE,
  flatten = FALSE,
  last = FALSE,
  limit = 0,
  withNa = TRUE,
  duplicatedValues = TRUE
)
```

Arguments:

`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.

`duplicatedValues` If set to TRUE, keeps duplicated values.

Returns: The value of the field.

Method `getFieldsByType()`**:** Gets the fields of this entry that have the specified type.

Usage:

```
BiodbEntry$getFieldsByType(type)
```

Arguments:

`type` The type of fields to retrieve.

Returns: A character vector containing the field names.

Method `getFieldsAsDataframe()`**:** Converts this entry into a data frame.

Usage:

```
BiodbEntry$getFieldsAsDataframe(
  only.atomic = TRUE,
  compute = TRUE,
  fields = NULL,
  fields.type = NULL,
  flatten = TRUE,
  limit = 0,
  only.card.one = FALSE,
  own.id = TRUE,
  duplicate.rows = TRUE,
  sort = FALSE,
  virtualFields = FALSE
)
```

Arguments:

`only.atomic` If set to TRUE, only export field's values that are atomic

`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.

(i.e. of type vector).

Returns: A data frame containg the values of the fields.

Method `getFieldsAsJson()`**:** Converts this entry into a JSON string.

Usage:

```
BiodbEntry$getFieldsAsJson(compute = TRUE)
```

Arguments:

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.

Returns: A JSON object from jsonlite package.

Method `parseContent()`**:** Parses content string and set values accordingly for this entry's fields. This method is called automatically and should be run directly by users.

Usage:

```
BiodbEntry$parseContent(content)
```

Arguments:

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.

Returns: Nothing.

Method `computeFields()`**:** Computes fields. Look at all missing fields, and try to compute them using references to other databases, if a rule exists.

Usage:

```
BiodbEntry$computeFields(fields = NULL)
```

Arguments:

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

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

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

Usage:

```
BiodbEntry$print()
```

Returns: Nothing.

Method `getName()`**:** Gets a short text describing this entry instance.

Usage:

```
BiodbEntry$getName()
```

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

Method `makesRefToEntry()`**:** Tests if this entry makes reference to another entry.

Usage:

```
BiodbEntry$makesRefToEntry(db, oid, recurse = FALSE)
```

Arguments:

`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`.

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

Method `getField()`**:** DEPRECATED. Gets the value of a field.

Usage:

```
BiodbEntry$getField(field)
```

Arguments:

`field` The name of the field.

Returns: The value of the field.

Method `setField()`**:** DEPRECATED. Sets the value of a field.

Usage:

```
BiodbEntry$setField(field, value)
```

Arguments:

`field` The name of the field.

`value` The new value of the field.

Returns: Nothing.

Method `getFieldClass()`**:** Gets the class of a field.

Usage:

```
BiodbEntry$getFieldClass(field)
```

Arguments:

`field` The name of the field.

Returns: The class of the field.

Method `getFieldDef()`**:** Gets the definition of an entry field.

Usage:

```
BiodbEntry$getFieldDef(field)
```

Arguments:

`field` The name of the field.

Returns: An object BiodbEntryField which defines the field.

Method `getFieldCardinality()`**:** Gets the cardinality of the field.

Usage:

```
BiodbEntry$getFieldCardinality(field)
```

Arguments:

`field` The name of the field.

Returns: The cardinality of the field.

Method `fieldHasBasicClass()`**:** DEPRECATED. Use BiodbEntryField::isVector() instead.

Usage:

```
BiodbEntry$fieldHasBasicClass(field)
```

Arguments:

`field` The name of the field.

Returns: TRUE if the field as a basic type (logical, numeric, character, ...).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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` *A class for describing an entry field.*

Description

A class for describing an entry field.

A class for describing an entry field.

Details

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.

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

Public methods:

- `BiodbEntryField$new()`
- `BiodbEntryField$getName()`
- `BiodbEntryField$getType()`
- `BiodbEntryField$isOfType()`
- `BiodbEntryField$getDescription()`
- `BiodbEntryField$hasAliases()`
- `BiodbEntryField$getAliases()`
- `BiodbEntryField$addAlias()`
- `BiodbEntryField$removeAlias()`
- `BiodbEntryField$getAllNames()`
- `BiodbEntryField$isComputable()`
- `BiodbEntryField$getComputableFrom()`

- BiodbEntryField$getDataFrameGroup()
- BiodbEntryField$isComputableFrom()
- BiodbEntryField$addComputableFrom()
- BiodbEntryField$removeComputableFrom()
- BiodbEntryField$correctValue()
- BiodbEntryField$isEnumerate()
- BiodbEntryField$isVirtual()
- BiodbEntryField$getVirtualGroupByType()
- BiodbEntryField$getAllowedValues()
- BiodbEntryField$addAllowedValue()
- BiodbEntryField$checkValue()
- BiodbEntryField$hasCardOne()
- BiodbEntryField$hasCardMany()
- BiodbEntryField$forbidsDuplicates()
- BiodbEntryField$isCaseInsensitive()
- BiodbEntryField$getClass()
- BiodbEntryField$isObject()
- BiodbEntryField$isDataFrame()
- BiodbEntryField$isAtomic()
- BiodbEntryField$isVector()
- BiodbEntryField$equals()
- BiodbEntryField$updateWithValuesFrom()
- BiodbEntryField$print()
- BiodbEntryField$getCardinality()
- BiodbEntryField$check()
- BiodbEntryField$clone()

Method `new()`**:** New instance initializer. This class must not be instantiated directly. Instead, you access the instances of this class through the BiodbEntryFields instance that you get from the BiodbMain instance.

Usage:

```
BiodbEntryField$new(
  parent,
  name,
  alias = NA_character_,
  type = NA_character_,
 class = c("character", "integer", "double", "logical", "object", "data.frame"),
  card = c("one", "many"),
  forbids.duplicates = FALSE,
  description = NA_character_,
  allowed.values = NULL,
  lower.case = FALSE,
  case.insensitive = FALSE,
  computable.from = NULL,
  virtual = FALSE,
  virtual.group.by.type = NULL,
  dataFrameGroup = NA_character_
)
```

Arguments:

`parent` The BiodbEntryFields parent instance.

`name` The field name.

`alias` The field aliases as a character vector.

`type` The field type.

`class` The field class.

`card` The field cardinality.

`forbids.duplicates` Set to TRUE to forbid duplicated values.

`description` The field description.

`allowed.values` Restrict possible values to a set of allowed values.

`lower.case` All values will be converted to lower case.

`case.insensitive` Comparison will be made case insensitive for this field.

`computable.from` A list of databases from which to compute automatically the value of this field.

`virtual` Set to TRUE if this field is virtual.

`virtual.group.by.type` In case of a virtual field, set the type of fields to group together into a data frame.

`dataFrameGroup` The data frame group.

Returns: Nothing.

Method `getName()`**:** Gets the name.

Usage:

```
BiodbEntryField$getName()
```

Returns: The name of this field.

Method `getType()`**:** Gets field's type.

Usage:

```
BiodbEntryField$getType()
```

Returns: The type of this field.

Method `isOfType()`**:** Tests if this field is of the specified type.

Usage:

```
BiodbEntryField$isOfType(type)
```

Arguments:

`type` The type.

Returns: TRUE if this field is of the specified type, FALSE otherwise.

Method `getDescription()`**:** Get field's description.

Usage:

```
BiodbEntryField$getDescription()
```

Returns: The description of this field.

Method `hasAliases()`**:** Tests if this field has aliases.

Usage:

```
BiodbEntryField$hasAliases()
```

Returns: TRUE if this entry field defines aliases, FALSE otherwise.

Method `getAliases()`**:** Get aliases.

Usage:

```
BiodbEntryField$getAliases()
```

Returns: The list of aliases if some are defined, otherwise returns NULL."

Method `addAlias()`**:** Adds an alias to the list of aliases.

Usage:

```
BiodbEntryField$addAlias(alias)
```

Arguments:

`alias` The name of a valid alias.

Returns: Nothing.

Method `removeAlias()`**:** Removes an alias from the list of aliases.

Usage:

```
BiodbEntryField$removeAlias(alias)
```

Arguments:

`alias` The name of a valid alias.

Returns: Nothing.

Method `getAllNames()`**:** Gets all names.

Usage:

```
BiodbEntryField$getAllNames()
```

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

Method `isComputable()`**:** Tests if this field is computable from another field or another database.

Usage:

```
BiodbEntryField$isComputable()
```

Returns: TRUE if the field is computable, FALSE otherwise.

Method `getComputableFrom()`**:** Get the list of connectors that can be used to compute this field.

Usage:

```
BiodbEntryField$getComputableFrom()
```

Returns: A list of list objects. Each list object contains the name of the database from which the field is computable.

Method `getDataFrameGroup()`**:** Gets the defined data frame group, if any.

Usage:

```
BiodbEntryField$getDataFrameGroup()
```

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

Method `isComputableFrom()`**:** Gets the ID of the database from which this field can be computed.

Usage:

```
BiodbEntryField$isComputableFrom()
```

Returns: The list of databases where to find this field's value.

Method `addComputableFrom()`**:** Adds a directive from the list of computableFrom.

Usage:

```
BiodbEntryField$addComputableFrom(directive)
```

Arguments:

`directive` A valid \"computable from\" directive.

Returns: Nothing.

Method `removeComputableFrom()`**:** Removes a directive from the list of computableFrom.

Usage:

```
BiodbEntryField$removeComputableFrom(directive)
```

Arguments:

`directive` A valid \"computable from\" directive.

Returns: Nothing.

Method `correctValue()`**:** Corrects a value so it is compatible with this field.

Usage:

```
BiodbEntryField$correctValue(value)
```

Arguments:

`value` A value.

Returns: The corrected value.

Method `isEnumerate()`**:** Tests if this field is an enumerate type (i.e.: it defines allowed values).

Usage:

```
BiodbEntryField$isEnumerate()
```

Returns: TRUE if this field defines some allowed values, FALSE otherwise.

Method `isVirtual()`**:** Tests if this field is a virtual field.

Usage:

```
BiodbEntryField$isVirtual()
```

Returns: TRUE if this field is virtual, FALSE otherwise.

Method `getVirtualGroupByType()`**:** Gets type for grouping field values when building a virtual data frame.

Usage:

```
BiodbEntryField$getVirtualGroupByType()
```

Returns: The type, as a character value.

Method `getAllowedValues()`**:** Gets allowed values.

Usage:

```
BiodbEntryField$getAllowedValues(value = NULL)
```

Arguments:

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

Returns: A character vector containing all allowed values.

Method `addAllowedValue()`: Adds an allowed value, as a synonym to already an existing value. Note that not all enumerate fields accept synonyms.

Usage:

```
BiodbEntryField$addAllowedValue(key, value)
```

Arguments:

`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.

Returns: Nothing.

Method `checkValue()`: Checks if a value is correct. Fails if `value` is incorrect.

Usage:

```
BiodbEntryField$checkValue(value)
```

Arguments:

`value` The value to check.

Returns: Nothing.

Method `hasCardOne()`: Tests if this field has a cardinality of one.

Usage:

```
BiodbEntryField$hasCardOne()
```

Returns: TRUE if the cardinality of this field is one, FALSE otherwise.

Method `hasCardMany()`: Tests if this field has a cardinality greater than one.

Usage:

```
BiodbEntryField$hasCardMany()
```

Returns: TRUE if the cardinality of this field is many, FALSE otherwise.

Method `forbidsDuplicates()`: Tests if this field forbids duplicates.

Usage:

```
BiodbEntryField$forbidsDuplicates()
```

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

Method `isCaseInsensitive()`: Tests if this field is case sensitive.

Usage:

```
BiodbEntryField$isCaseInsensitive()
```

Returns: TRUE if this field is case insensitive, FALSE otherwise.

Method `getClass()`: Gets the class of this field's value.

Usage:

```
BiodbEntryField$getClass()
```

Returns: class) of this field.

Method `isObject()`: Tests if this field's type is a class.

Usage:

```
BiodbEntryField$isObject()
```

Returns: TRUE if field's type is a class, FALSE otherwise.

Method `isDataFrame()`**:** Tests if this field's type is `data.frame`.

Usage:

```
BiodbEntryField$isDataFrame()
```

Returns: TRUE if field's type is data frame, FALSE otherwise."

Method `isAtomic()`**:** Tests if this field's type is an atomic type.

Usage:

```
BiodbEntryField$isAtomic()
```

Returns: character, integer, double or logical), FALSE otherwise.

Method `isVector()`**:** Tests if this field's type is a basic vector type.

Usage:

```
BiodbEntryField$isVector()
```

Returns: character, integer, double or logical), FALSE otherwise.

Method `equals()`**:** Compares this instance with another, and tests if they are equal.

Usage:

```
BiodbEntryField$equals(other, fail = FALSE)
```

Arguments:

`other` Another BiodbEntryField instance.

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

Returns: TRUE if they are equal, FALSE otherwise.

Method `updateWithValuesFrom()`**:** Updates fields using values from `other` instance. The updated fields

Usage:

```
BiodbEntryField$updateWithValuesFrom(other)
```

Arguments:

`other` Another BiodbEntryField instance.

`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.

Returns: Nothing.

Method `print()`**:** Print informations about this entry.

Usage:

```
BiodbEntryField$print()
```

Returns: Nothing.

Method `getCardinality()`**:** Gets the field's cardinality.

Usage:

```
BiodbEntryField$getCardinality()
```

Returns: The cardinality: "one" or "many".

Method `check()`**:** Checks if essential values are defined.

Usage:

`BiodbEntryField$check()`

Returns: Nothing.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

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 *A class for handling description of all entry fields.*

Description

A class for handling description of all entry fields.

A class for handling description of all entry fields.

Details

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

Methods

Public methods:

- `BiodbEntryFields$new()`
- `BiodbEntryFields$notifyCfgUpdate()`
- `BiodbEntryFields$isAlias()`
- `BiodbEntryFields$formatName()`

- BiodbEntryFields$isDefined()
- BiodbEntryFields$checkIsDefined()
- BiodbEntryFields$getRealName()
- BiodbEntryFields$get()
- BiodbEntryFields$getFieldNames()
- BiodbEntryFields$getDatabaseIdField()
- BiodbEntryFields$print()
- BiodbEntryFields$define()
- BiodbEntryFields$terminate()
- BiodbEntryFields$clone()

Method `new()`**:** New instance initializer. No BiodbEntryFields instance must be created directly. Instead, call the getEntryFields() method of BiodbMain.

Usage:

```
BiodbEntryFields$new(parent)
```

Arguments:

`parent` The BiodbMain instance.

Returns: Nothing.

Method `notifyCfgUpdate()`**:** Call back method called when a value is modified inside the configuration.

Usage:

```
BiodbEntryFields$notifyCfgUpdate(k, v)
```

Arguments:

`k` The config key name.

`v` The value associated with the key.

Returns: Nothing.

Method `isAlias()`**:** Tests if names are aliases.

Usage:

```
BiodbEntryFields$isAlias(name)
```

Arguments:

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

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

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

Usage:

```
BiodbEntryFields$formatName(name)
```

Arguments:

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

Returns: A character vector of formatted names.

Method `isDefined()`**:** Tests if names are defined fields.

Usage:

```
BiodbEntryFields$isDefined(name)
```

Arguments:

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

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

Method `checkIsDefined()`**:** Tests if names are valid defined fields. Throws an error if any name does not correspond to a defined field.

Usage:

```
BiodbEntryFields$checkIsDefined(name)
```

Arguments:

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

Returns: Nothing.

Method `getRealName()`**:** 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.

Usage:

```
BiodbEntryFields$getRealName(name, fail = TRUE)
```

Arguments:

`name` A character vector of names or aliases.

`fail` Fails if name is unknown.

Returns: 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).

Method `get()`**:** Gets a BiodbEntryField instance.

Usage:

```
BiodbEntryFields$get(name, drop = TRUE)
```

Arguments:

`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.

Returns: 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.

Method `getFieldNames()`**:** Gets the main names of all fields.

Usage:

```
BiodbEntryFields$getFieldNames(type = NULL, computable = NULL)
```

Arguments:

`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.

Returns: A character vector containing all selected field names.

Method `getDatabaseIdField()`**:** Gets a database ID field.

Usage:

`BiodbEntryFields$getDatabaseIdField(database)`

Arguments:

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

Returns:

`accession numbers) for this database.`

Method `print()`**:** Prints information about the instance.

Usage:

`BiodbEntryFields$print()`

Returns: Nothing.

Method `define()`**:** Defines fields.

Usage:

`BiodbEntryFields$define(def)`

Arguments:

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

Returns: Nothing.

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

Usage:

`BiodbEntryFields$terminate()`

Arguments:

`BiodbMain` :terminate().

Returns: Nothing.

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

Usage:

`BiodbEntryFields$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

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` *A class for constructing biodb objects.*

Description

A class for constructing biodb objects.

A class for constructing biodb objects.

Details

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

Public methods:

- `BiodbFactory$new()`
- `BiodbFactory$getBiodb()`
- `BiodbFactory$createConn()`
- `BiodbFactory$connExists()`
- `BiodbFactory$deleteConn()`
- `BiodbFactory$deleteConnByClass()`
- `BiodbFactory$getAllConnectors()`
- `BiodbFactory$deleteAllConnectors()`
- `BiodbFactory$getConn()`
- `BiodbFactory$getEntry()`
- `BiodbFactory$createNewEntry()`
- `BiodbFactory$createEntryFromContent()`
- `BiodbFactory$getAllCacheEntries()`
- `BiodbFactory$deleteAllEntriesFromVolatileCache()`
- `BiodbFactory$deleteAllCacheEntries()`
- `BiodbFactory$print()`
- `BiodbFactory$clone()`

Method `new()`**:** New instance initializer. The BiodbFactory class must not be instantiated directly. Instead, call the getFactory() method from the BiodbMain instance.

Usage:

```
BiodbFactory$new(bdb)
```

Arguments:

`bdb` The BiodbMain instance.

Returns: Nothing.

Method `getBiodb()`**:** Returns the biodb main class instance to which this object is attached.

Usage:

```
BiodbFactory$getBiodb()
```

Returns: The main biodb instance.

Method `createConn()`**:** Creates a connector to a database.

Usage:

```
BiodbFactory$createConn(
  db.class,
  url = NULL,
  token = NA_character_,
  fail.if.exists = TRUE,
  get.existing.conn = TRUE,
  conn.id = NULL,
  cache.id = NULL
)
```

Arguments:

`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.

Returns: An instance of the requested connector class.

Method `connExists()`**:** Tests if a connector exists.

Usage:

```
BiodbFactory$connExists(conn.id)
```

Arguments:

`conn.id` A connector ID.

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

Method `deleteConn()`**:** Deletes an existing connector.

Usage:

```
BiodbFactory$deleteConn(conn)
```

Arguments:

`conn` A connector instance or a connector ID.

Returns: Nothing.

Method `deleteConnByClass()`**:** Deletes all existing connectors from a same class.

Usage:

```
BiodbFactory$deleteConnByClass(db.class)
```

Arguments:

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

Returns: Nothing.

Method `getAllConnectors()`**:** Gets all connectors.

Usage:

```
BiodbFactory$getAllConnectors()
```

Returns: A list of all created connectors.

Method `deleteAllConnectors()`**:** Deletes all connectors.

Usage:

```
BiodbFactory$deleteAllConnectors()
```

Returns: Nothing.

Method `getConn()`**:** Gets an instantiated connector instance, or create a new one.

Usage:

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

Arguments:

`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\".

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

Method `getEntry()`**:** Retrieves database entry objects from IDs (accession numbers), for the specified connector.

Usage:

```
BiodbFactory$getEntry(conn.id, id, drop = TRUE, no.null = FALSE, limit = 0)
```

Arguments:

`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.

`no.null` Set to TRUE to remove NULL entries.

`limit` Set to a positive value to limit the number of entries returned.

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

Method `createNewEntry()`**:** Creates a new empty entry object from scratch. This entry is not stored in cache, and is directly attached to the factory instance instead of a particular connector.

Usage:

```
BiodbFactory$createNewEntry(db.class)
```

Arguments:

`db.class` A database ID.

Returns: A new BiodbEntry object.

Method `createEntryFromContent()`**:** Creates an entry instance from a content.

Usage:

```
BiodbFactory$createEntryFromContent(conn.id, content, drop = TRUE)
```

Arguments:

`conn.id` A valid BiodbConn identifier.

`content` A list or character vector of contents to parse to create the entries.

`drop` If set to TRUE

Returns: A list of new BiodbEntry objects.

Method `getAllCacheEntries()`**:** For a connector, gets all entries stored in the cache.

Usage:

```
BiodbFactory$getAllCacheEntries(conn.id)
```

Arguments:

`conn.id` A connector ID.

Returns: A list of BiodbEntry objects.

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

Usage:

```
BiodbFactory$deleteAllEntriesFromVolatileCache(conn.id)
```

Arguments:

`conn.id` A connector ID.

Returns: Nothing.

Method `deleteAllCacheEntries()`**:** Deletes all entries stored in the cache of the given connector.

Usage:

```
BiodbFactory$deleteAllCacheEntries(conn.id)
```

Arguments:

`conn.id` A connector ID.

Returns: Nothing.

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

Usage:

```
BiodbFactory$print()
```

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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 *Entry class for content in HTML format.*

Description

Entry class for content in HTML format.

Entry class for content in HTML format.

Details

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

Super classes

biodb::BiodbEntry -> biodb::BiodbXmlEntry -> BiodbHtmlEntry

Methods

Public methods:

- BiodbHtmlEntry$new()
- BiodbHtmlEntry$clone()

Method `new()`**:** New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
BiodbHtmlEntry$new(...)
```

Arguments:

... All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class `BiodbXmlEntry`.

Examples

```
# Create a concrete entry class inheriting from this class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbHtmlEntry)
```

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

Description

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

Super class

`biodb::BiodbEntry` -> `BiodbJsonEntry`

Methods

Public methods:

- `BiodbJsonEntry$clone()`

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class `BiodbEntry`.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbJsonEntry)
```

BiodbListEntry *Entry class for content in list format.*

Description

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

Super class

biodb::BiodbEntry -> BiodbListEntry

Methods

Public methods:

- BiodbListEntry$clone()

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbListEntry)
```

BiodbMain *The central class of the biodb package.*

Description

The central class of the biodb package.

The central class of the biodb package.

Details

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

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, BiodbConfig, ...) are instantiated (just once) and their instances are later accessible through get*() methods.

Methods

Public methods:

- BiodbMain$new()
- BiodbMain$terminate()
- BiodbMain$loadDefinitions()
- BiodbMain$getConfig()
- BiodbMain$getPersistentCache()
- BiodbMain$getDbsInfo()
- BiodbMain$getEntryFields()
- BiodbMain$getFactory()
- BiodbMain$getRequestScheduler()
- BiodbMain$addObservers()
- BiodbMain$getObservers()
- BiodbMain$convertEntryIdFieldToDbClass()
- BiodbMain$entriesFieldToVctOrLst()
- BiodbMain$entriesToDataframe()
- BiodbMain$entryIdsToDataframe()
- BiodbMain$addColsToDataframe()
- BiodbMain$entriesToJson()
- BiodbMain$collapseRows()
- BiodbMain$entriesToSingleFieldValues()
- BiodbMain$entryIdsToSingleFieldValues()
- BiodbMain$computeFields()
- BiodbMain$saveEntriesAsJson()
- BiodbMain$copyDb()
- BiodbMain$print()
- BiodbMain$fieldIsAtomic()
- BiodbMain$getFieldClass()
- BiodbMain$clone()

Method `new()`**:** New instance initializer. The BiodbMain must not be instantiated directly. Instead use the newInst() global method.

Usage:

```
BiodbMain$new(autoloadExtraPkgs = NULL)
```

Arguments:

`autoloadExtraPkgs` Set to TRUE to allow automatic loading of extension packages. Set to FALSE to forbid it. If left to NULL, the default, autoload.extra.pkgs configuration value will be used.

Returns: Nothing.

Method `terminate()`**:** Closes `BiodbMain` instance. Call this method when you are done with your `BiodbMain` instance.

Usage:

```
BiodbMain$terminate()
```

Returns: Nothing.

Method `loadDefinitions()`**:** Loads databases and entry fields definitions from YAML file.

Usage:

```
BiodbMain$loadDefinitions(file, package = "biodb")
```

Arguments:

`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.

Returns: Nothing.

Method `getConfig()`**:** Returns the single instance of the `BiodbConfig` class.

Usage:

```
BiodbMain$getConfig()
```

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

Method `getPersistentCache()`**:** Returns the single instance of the BiodbPersistentCache class.

Usage:

```
BiodbMain$getPersistentCache()
```

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

Method `getDbsInfo()`**:** Returns the single instance of the `BiodbDbsInfo` class.

Usage:

```
BiodbMain$getDbsInfo()
```

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

Method `getEntryFields()`**:** Returns the single instance of the `BiodbEntryFields` class.

Usage:

```
BiodbMain$getEntryFields()
```

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

Method `getFactory()`**:** Returns the single instance of the `BiodbFactory` class.

Usage:

```
BiodbMain$getFactory()
```

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

Method `getRequestScheduler()`**:** Returns the single instance of the Scheduler class.

Usage:

```
BiodbMain$getRequestScheduler()
```

Returns: The instance of the Scheduler class attached to this BiodbMain instance.

Method `addObservers()`**:** 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.

Usage:

```
BiodbMain$addObservers(observers)
```

Arguments:

`observers` Either an object or a list of objects.

Returns: Nothing.

Method `getObservers()`**:** Gets the list of registered observers.

Usage:

```
BiodbMain$getObservers()
```

Returns: The list or registered observers.

Method `convertEntryIdFieldToDbClass()`**:** Gets the database class name corresponding to an entry ID field.

Usage:

```
BiodbMain$convertEntryIdFieldToDbClass(entry.id.field)
```

Arguments:

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

Method `entriesFieldToVctOrLst()`**:** 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.

Usage:

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

Arguments:

`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.

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

Method `entriesToDataframe()`**:** Converts a list of entries or a list of list of entries (`BiodbEntry` objects) into a data frame.

Usage:

```
BiodbMain$entriesToDataframe(
  entries,
  only.atomic = TRUE,
  null.to.na = TRUE,
  compute = TRUE,
  fields = NULL,
  limit = 0,
  drop = FALSE,
  sort.cols = FALSE,
  flatten = TRUE,
  only.card.one = FALSE,
```

```
  own.id = TRUE,
  prefix = ""
)
```

Arguments:

`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.

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

Method `entryIdsToDataframe()`**:** Construct a data frame using entry IDs and field values of the corresponding entries.

Usage:

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

Arguments:

`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.

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

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

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

Method `addColsToDataframe()`**:** Add values from a database to an existing data frame using a column containing entry identifiers.

Usage:

```
BiodbMain$addColsToDataframe(x, id.col, db, fields, limit = 3, prefix = "")
```

Arguments:

`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 single 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.

Returns: A data frame containing `x` and new columns appended for the fields requested.

Method `entriesToJson()`**:** Converts a list of `BiodbEntry` objects into JSON. Returns a vector of characters.

Usage:

```
BiodbMain$entriesToJson(entries, compute = TRUE)
```

Arguments:

`entries` A list of `BiodbEntry` instances. It may contain NULL elements.

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

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

Method `collapseRows()`**:** 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.

Usage:

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

Arguments:

`x` A data frame.

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

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

Returns: A data frame, with rows collapsed."

Method `entriesToSingleFieldValues()`**:** Extract all values of a field from a list of entries.

Usage:

```
BiodbMain$entriesToSingleFieldValues(
  entries,
  field,
  sortOutput = FALSE,
  uniq = TRUE
)
```

Arguments:

`entries` A list of BiodbEntry objects.

`field` The field for which to extract values.

`sortOutput` Set to TRUE to sort the values.

`uniq` Set to TRUE to remove duplicates.

Returns: The values of the field as a vector.

Method `entryIdsToSingleFieldValues()`**:** Extract all values of a field from a list of entries.

Usage:

```
BiodbMain$entryIdsToSingleFieldValues(
  ids,
  db,
  field,
  sortOutput = FALSE,
  uniq = TRUE
)
```

Arguments:

`ids` A list of entry identifiers.

`db` The database ID or connector ID where to find the entries.

`field` The field for which to extract values.

`sortOutput` Set to TRUE to sort the values.

`uniq` Set to TRUE to remove duplicates.

Returns: The values of the field as a vector.

Method `computeFields()`**:** Computes missing fields in entries, for those fields that are comptable.

Usage:

```
BiodbMain$computeFields(entries)
```

Arguments:

`entries` A list of `BiodbEntry` instances. It may contain NULL elements.

Returns: Nothing.

Method `saveEntriesAsJson()`**:** Saves a list of entries in JSON format. Each entry will be saved in a separate file.

Usage:

```
BiodbMain$saveEntriesAsJson(entries, files, compute = TRUE)
```

Arguments:

`entries` A list of `BiodbEntry` instances. It may contain NULL elements.

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

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

Returns: Nothing.

Method `copyDb()`**:** Copies all entries of a database into another database. The connector of the destination database must be editable.

Usage:

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

Arguments:

`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.

Returns: Nothing.

Method `print()`**:** Prints object information.

Usage:

```
BiodbMain$print()
```

Returns: Nothing.

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

Usage:

```
BiodbMain$fieldIsAtomic(field)
```

Arguments:

`field` The name of the field.

Returns: TRUE if the field's value is atomic.

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

Usage:

```
BiodbMain$getFieldClass(field)
```

Arguments:

`field` The name of the field.

Returns: The class of the field.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

`BiodbFactory`, `BiodbConfig`, `BiodbEntryFields`, `BiodbDbsInfo`.

Examples

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

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

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

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

Description

Entry class for content in SDF format.

Entry class for content in SDF format.

Details

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

Super classes

`biodb::BiodbEntry` -> `biodb::BiodbTxtEntry` -> `BiodbSdfEntry`

Methods

Public methods:

- `BiodbSdfEntry$new()`
- `BiodbSdfEntry$clone()`

Method `new()`**:** New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
BiodbSdfEntry$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Super class `BiodbTxtEntry`.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbSdfEntry)
```

BiodbTestMsgAck *A class for acknowledging messages during tests.*

Description

A class for acknowledging messages during tests.

A class for acknowledging messages during tests.

Details

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

Public methods:

- `BiodbTestMsgAck$new()`
- `BiodbTestMsgAck$notifyProgress()`
- `BiodbTestMsgAck$clone()`

Method `new()`**:** New instance initializer.

Usage:

```
BiodbTestMsgAck$new()
```

Returns: Nothing.

Method `notifyProgress()`**:** Call back method used to get progress advancement of a long process.

Usage:

```
BiodbTestMsgAck$notifyProgress(what, index, total)
```

Arguments:

`what` The reason as a character value.

`index` The index number representing the progress.

`total` The total number to reach for completing the process.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

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

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

BiodbTxtEntry *Entry class for content in text format.*

Description

Entry class for content in text format.

Entry class for content in text format.

Details

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

Super class

biodb::BiodbEntry -> BiodbTxtEntry

Methods

Public methods:

- BiodbTxtEntry$new()
- BiodbTxtEntry$clone()

Method `new()`**:** New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
BiodbTxtEntry$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbTxtEntry)
```

BiodbXmlEntry *Entry class for content in XML format.*

Description

Entry class for content in XML format.

Entry class for content in XML format.

Details

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

Super class

biodb::BiodbEntry -> BiodbXmlEntry

Methods

Public methods:

- BiodbXmlEntry$new()
- BiodbXmlEntry$clone()

Method new(): New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
BiodbXmlEntry$new(...)
```

Arguments:

... All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class BiodbEntry.

Examples

```
# Create a concrete entry class inheriting from CSV class:
MyEntry <- R6::R6Class("MyEntry", inherit=biodb::BiodbXmlEntry)
```

checkDeprecatedCacheFolders *Check deprecated default cache folders.*

Description

Searches for a deprecated location of the default cache folder, and moves files to the new location if possible. Otherwise raises a warning.

Usage

```
checkDeprecatedCacheFolders()
```

Value

Nothing.

Examples

```
biodb::checkDeprecatedCacheFolders()
```

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 *Compound CSV File connector class.*

Description

Compound CSV File connector class.

Compound CSV File connector class.

Details

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

Super classes

biodb::BiodbConnBase -> biodb::BiodbConn -> biodb::CsvFileConn -> CompCsvFileConn

Methods

Public methods:

- CompCsvFileConn$new()
- CompCsvFileConn$clone()

Method `new()`**:** New instance initializer. Connector classes must not be instantiated directly. Instead, you must use the createConn() method of the factory class.

Usage:

```
CompCsvFileConn$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Super class CsvFileConn.

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` *Compound CSV File entry class.*

Description

Compound CSV File entry class.

Compound CSV File entry class.

Details

This is the entry class for Compound CSV file databases.

Super classes

`biodb::BiodbEntry` -> `biodb::BiodbCsvEntry` -> `CompCsvFileEntry`

Methods

Public methods:

- `CompCsvFileEntry$new()`
- `CompCsvFileEntry$clone()`

Method `new()`**:** New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
CompCsvFileEntry$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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 for handling a Compound database in SQLite format.*

Description

This is the connector class for a Compound database.

Super classes

biodb::BiodbConnBase -> biodb::BiodbConn -> biodb::SqliteConn -> CompSqliteConn

Methods

Public methods:

- CompSqliteConn$clone()

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class SqliteConn.

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 *Compound SQLite entry class.*

Description

This is the entry class for a Compound SQLite database.

Super classes

biodb::BiodbEntry -> biodb::BiodbListEntry -> CompSqliteEntry

Methods

Public methods:

- CompSqliteEntry$clone()

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

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` *CSV File connector class.*

Description

CSV File connector class.

CSV File connector class.

Details

This is the abstract connector class for all CSV file databases.

Super classes

`biodb::BiodbConnBase` -> `biodb::BiodbConn` -> `CsvFileConn`

Methods

Public methods:

- `CsvFileConn$new()`
- `CsvFileConn$getCsvQuote()`
- `CsvFileConn$setCsvQuote()`
- `CsvFileConn$getCsvSep()`
- `CsvFileConn$setCsvSep()`
- `CsvFileConn$getFieldNames()`
- `CsvFileConn$hasField()`
- `CsvFileConn$addField()`
- `CsvFileConn$getFieldColName()`
- `CsvFileConn$setField()`
- `CsvFileConn$getFieldsAndColumnsAssociation()`
- `CsvFileConn$getUnassociatedColumns()`
- `CsvFileConn$print()`
- `CsvFileConn$setDb()`
- `CsvFileConn$setIgnoreUnassignedColumns()`
- `CsvFileConn$clone()`

Method `new()`**:** New instance initializer. Connector classes must not be instantiated directly. Instead, you must use the createConn() method of the factory class.

Usage:

```
CsvFileConn$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

Method `getCsvQuote()`**:** Gets the characters used to delimit quotes in the CSV database file.

Usage:

```
CsvFileConn$getCsvQuote()
```

Returns: The characters used to delimit quotes as a single character value.

Method `setCsvQuote()`**:** Sets the characters used to delimit quotes in the CSV database file.

Usage:

```
CsvFileConn$setCsvQuote(quote)
```

Arguments:

`quote` The characters used to delimit quotes as a single character value.
`You` may specify several characters. Example \"\\"'\".

Returns: Nothing.

Method `getCsvSep()`**:** Gets the current CSV separator used for the database file.

Usage:

```
CsvFileConn$getCsvSep()
```

Returns: The CSV separator as a character value.

Method `setCsvSep()`**:** 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.

Usage:

```
CsvFileConn$setCsvSep(sep)
```

Arguments:

`sep` The CSV separator as a character value.

Returns: Nothing.

Method `getFieldNames()`**:** Get the list of all biodb fields handled by this database.

Usage:

```
CsvFileConn$getFieldNames()
```

Returns: A character vector of the biodb field names.

Method `hasField()`**:** Tests if a field is defined for this database instance.

Usage:

```
CsvFileConn$hasField(field)
```

Arguments:

`field` A valid Biodb entry field name.

Returns: TRUE of the field is defined, FALSE otherwise.

Method `addField()`**:** 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.

Usage:

```
CsvFileConn$addField(field, value)
```

Arguments:

`field` A valid Biodb entry field name.
`value` The value to set for this field.

Returns: Nothing.

Method `getFieldColName()`**:** Get the column name corresponding to a Biodb field.

Usage:

```
CsvFileConn$getFieldColName(field)
```

Arguments:

`field` A valid Biodb entry field name. This field must be defined for this database instance.

Returns: The column name from the CSV file.

Method `setField()`**:** Sets a field by making a correspondence between a Biodb field and one or more columns of the loaded data frame.

Usage:

```
CsvFileConn$setField(field, colname, ignore.if.missing = FALSE)
```

Arguments:

`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.

Returns: Nothing.

Method `getFieldsAndColumnsAssociation()`**:** Gets the association between biodb field names and CSV file column names.

Usage:

```
CsvFileConn$getFieldsAndColumnsAssociation()
```

Returns: A list with names being the biodb field names and values being a character vector of column names from the CSV file.

Method `getUnassociatedColumns()`**:** Gets the list of unassociated column names from the CSV file.

Usage:

```
CsvFileConn$getUnassociatedColumns()
```

Returns: A character vector containing column names.

Method `print()`**:** Prints a description of this connector.

Usage:

```
CsvFileConn$print()
```

Returns: Nothing.

Method `setDb()`**:** Sets the database directly from a data frame. You must not have set the database previously with the URL parameter.

Usage:

```
CsvFileConn$setDb(db)
```

Arguments:

`db` A data frame containing your database.

Returns: Nothing.

Method `setIgnoreUnassignedColumns()`**:** Tells the connector to ignore or not the columns found in the CSV file for which no assignment were found.

Usage:

`CsvFileConn$setIgnoreUnassignedColumns(ignore)`

Arguments:

`ignore` Set to TRUE to ignore the unassigned columns, and to FALSE otherwise.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Super classes `BiodbConn`, 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})
```

`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()`**:** Initializer.

Usage:

```
FileTemplate$new(path)
```

Arguments:

`path` The path to the template file.

Returns: Nothing.

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: Nothing.

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.

`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()
```

---

getDefaultCacheDir *Get default cache folder.*

Description

Returns the path to the default cache folder.

Usage

```
getDefaultCacheDir()
```

Value

The path to the cache folder.

Examples

```
cacheFolderPath <- biodb::getDefaultCacheDir()
```

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()
```

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()
```

listTestRefEntries *List test reference entries.*

Description

DEPRECATED. Use TestRefEntries class instead.

Usage

```
listTestRefEntries(conn.id, pkgName, limit = 0)
```

Arguments

conn.id	A valid Biodb connector ID.
pkgName	The name of the
limit	The maximum number of entries to retrieve.

Details

Lists the reference entries in the test folder for a specified connector. The test reference files must be in <pkg>/inst/testref/ folder and their names must match entry-<database_name>-<entry_accession>.json (e.g.: entry-comp.csv.file-1018.json).

Value

A list of entry IDs.

Examples

```
# List IDs of test reference entries:
biodb::listTestRefEntries('comp.csv.file', pkgName='biodb')
```

loadFileContents *Loads the contents of files in memory.*

Description

This function loads the contents of a list of files and returns the contents as a list, each element being the content of a single file, in the same order. If a file could not be opened, a NULL value is used as the content. NA values are interpreted by default, but this behaviour can be turned off.

Usage

```
loadFileContents(x, naValues = "NA", outVect = FALSE)
```

Arguments

x	A character vector containing the paths of the files.
naValues	A character vector listing the content values to convert into NA value. Set to NULL to disable the interpretation of NA values. set to a different set of values to be interpreted.
outVect	If set to TRUE outputs a character vector (converting any NULL value into NA), otherwise outputs a list.

Value

A list with the contents of the files.

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 *Mass CSV File connector class.*

Description

Mass CSV File connector class.

Mass CSV File connector class.

Details

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

Super classes

`biodb::BiodbConnBase` -> `biodb::BiodbConn` -> `biodb::CsvFileConn` -> `MassCsvFileConn`

Methods

Public methods:

- `MassCsvFileConn$new()`
- `MassCsvFileConn$getPrecursorFormulae()`
- `MassCsvFileConn$isAPrecursorFormula()`
- `MassCsvFileConn$setPrecursorFormulae()`
- `MassCsvFileConn$addPrecursorFormulae()`
- `MassCsvFileConn$clone()`

Method `new()`**:** New instance initializer. Connector classes must not be instantiated directly. Instead, you must use the createConn() method of the factory class.

Usage:

```
MassCsvFileConn$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

Method `getPrecursorFormulae()`**:** Gets the list of formulae used to recognize precursors.

Usage:

```
MassCsvFileConn$getPrecursorFormulae()
```

Returns: A character vector containing chemical formulae.

Method `isAPrecursorFormula()`**:** Tests if a formula is a precursor formula.

Usage:

```
MassCsvFileConn$isAPrecursorFormula(formula)
```

Arguments:

`formula` A chemical formula, as a character value.

Returns: TRUE if the submitted formula is considered a precursor.

Method `setPrecursorFormulae()`**:** Sets the list precursor formulae.

Usage:

```
MassCsvFileConn$setPrecursorFormulae(formulae)
```

Arguments:

`formulae` A character vector containing formulae.

Returns: Nothing.

Method `addPrecursorFormulae()`**:** Adds new formulae to the list of formulae used to recognize precursors.

Usage:

```
MassCsvFileConn$addPrecursorFormulae(formulae)
```

Arguments:

`formulae` A character vector containing formulae.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Super class `CsvFileConn`.

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 *Mass CSV File entry class.*

Description

Mass CSV File entry class.

Mass CSV File entry class.

Details

This is the entry class for Mass CSV file databases.

Super classes

biodb::BiodbEntry -> biodb::BiodbCsvEntry -> MassCsvFileEntry

Methods

Public methods:

- MassCsvFileEntry$new()
- MassCsvFileEntry$clone()

Method `new()`**:** New instance initializer. Entry objects must not be created directly. Instead, they are retrieved through the connector instances.

Usage:

```
MassCsvFileEntry$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

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 for handling a Mass spectrometry database in SQLite format.*

Description

This is the connector class for a MASS SQLite database.

Super classes

biodb::BiodbConnBase -> biodb::BiodbConn -> biodb::SqliteConn -> MassSqliteConn

Methods

Public methods:

- MassSqliteConn$clone()

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class 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 *Mass spectra SQLite entry class.*

Description

This is the entry class for a Mass spectra SQLite database.

Super classes

biodb::BiodbEntry -> biodb::BiodbListEntry -> MassSqliteEntry

Methods

Public methods:

- MassSqliteEntry$clone()

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

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()`**:** Initializer.

Usage:

```
Progress$new(biodb = NULL, msg, total = NA_integer_)
```

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 or NA if unknown.

Returns: Nothing.

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=10L)

# 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$getPpm()`
- `Range$getTolExpr()`
- `Range$clone()`

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

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: Nothing.

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 `getTolExpr()`**:** Gets the tolerance expression as a list.

Usage:

```
Range$getTolExpr()
```

Returns: A list containing the tolerance range expression.

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,
  pkgName,
  testRefFolder = NULL,
  opt = NULL,
  short = TRUE,
  long = FALSE,
  maxShortTestRefEntries = 1
)
```

Arguments

`conn`	A valid biodb connector.
`pkgName`	The name of your package.
`testRefFolder`	The folder where to find test reference files.
`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
## Not run:
biodb::runGenericTests(conn, 'mypkg')

## End(Not run)

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

SqliteConn *SQLite connector class.*

Description

SQLite connector class.

SQLite connector class.

Details

This is the abstract connector class for all SQLite databases.

Super classes

`biodb::BiodbConnBase` -> `biodb::BiodbConn` -> `SqliteConn`

Methods

Public methods:

- `SqliteConn$new()`
- `SqliteConn$hasField()`
- `SqliteConn$getQuery()`
- `SqliteConn$clone()`

Method `new()`**:** New instance initializer. Connector classes must not be instantiated directly. Instead, you must use the createConn() method of the factory class.

Usage:

```
SqliteConn$new(...)
```

Arguments:

`...` All parameters are passed to the super class initializer.

Returns: Nothing.

Method `hasField()`**:** Tests if a field is defined for this database instance.

Usage:

```
SqliteConn$hasField(field)
```

Arguments:

`field` A valid Biodb entry field name.

Returns: TRUE of the field is defined, FALSE otherwise.

Method `getQuery()`**:** Run a query using a biodb SQL object.

Usage:

```
SqliteConn$getQuery(query)
```

Arguments:

`query` A valid BiodbSqlQuery object.

Returns: The result returned by DBI::dbGetQuery() call.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Super class BiodbConn 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()
```

`TestRefEntries` *A class for accessing the test reference entries.*

Description

A class for accessing the test reference entries.

A class for accessing the test reference entries.

Details

The test reference entries are stored as JSON files inside `inst/testref` folder of each extension package.

Methods

Public methods:

- `TestRefEntries$new()`
- `TestRefEntries$getAllIds()`
- `TestRefEntries$getContents()`
- `TestRefEntries$getRealEntries()`
- `TestRefEntries$saveEntriesAsJson()`
- `TestRefEntries$getRealEntry()`
- `TestRefEntries$getRefEntry()`
- `TestRefEntries$getAllRefEntriesDf()`
- `TestRefEntries$clone()`

Method `new()`**:** New instance initializer.

Usage:

```
TestRefEntries$new(db.class, pkgName, folder = NULL, bdb = NULL)
```

Arguments:

`db.class` Identifier of the database.

`pkgName` Name of the package in which are stored the reference entry files.

`folder` The folder where to find test reference files for the package. Usually it is "inst/testref".

`bdb` A valid BiodbMain instance or NULL.

Returns: Nothing.

Method `getAllIds()`**:** Retrieve all identifiers.

Usage:

```
TestRefEntries$getAllIds(limit = 0)
```

Arguments:

`limit` The maximum number of identifiers to return.

Returns: A character vector containing the IDs.

Method `getContents()`**:** Get the reference contents for the specfied IDs.

Usage:

```
TestRefEntries$getContents(ids)
```

Arguments:

`ids` The reference IDs for which to get the contents.

Returns: A character vector.

Method `getRealEntries()`**:** Retrieve all real entries from database corresponding to the reference entris.

Usage:

```
TestRefEntries$getRealEntries(ids = NULL)
```

Arguments:

`ids` A character vector of entry identifiers.

Returns: A list containing BiodbEntry instances.

Method `saveEntriesAsJson()`**:** Saves a list of entries into separate JSON files, inside the test output folder.

Usage:

```
TestRefEntries$saveEntriesAsJson(ids, entries)
```

Arguments:

`ids` The IDs of the entries.

`entries` A list of entries. It can contain NULL values.

Returns: Nothing.

Method `getRealEntry()`**:** Retrieves one real entry from database corresponding to one reference entry.

Usage:

```
TestRefEntries$getRealEntry(id)
```

Arguments:

`id` The identifier of the entry.

Returns: A BiodbEntry instance.

Method `getRefEntry()`**:** Retrieves the content of a single reference entry.

Usage:

```
TestRefEntries$getRefEntry(id)
```

Arguments:

`id` The identifier of the reference entry to retrieve.

Returns: The content of the reference entry as a list.

Method `getAllRefEntriesDf()`**:** Load all reference entries.

Usage:

```
TestRefEntries$getAllRefEntriesDf()
```

Returns: A data frame containing all the reference entries with their values.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Creates an instance
refEntries <- TestRefEntries$new('comp.sqlite', pkgName='biodb')

# Gets identifiers of all reference entries
refEntries$getAllIds()

# Gets a data frame with the content of the reference entries
refEntries$getAllRefEntriesDf()
```

`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()
```

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

abstractClass, 4
abstractMethod, 4

biodb *(biodb-package)*, 3
biodb-package, 3
biodb::BiodbConn, *76*, *78*, *81*, *95*, *98*, *105*
biodb::BiodbConnBase, *10*, *35*, *76*, *78*, *81*, *95*, *98*, *105*
biodb::BiodbCsvEntry, *77*, *97*
biodb::BiodbEntry, *35*, *61–63*, *71*, *73, 74*, *77*, *79*, *97*, *99*
biodb::BiodbListEntry, *79*, *99*
biodb::BiodbTxtEntry, *71*
biodb::BiodbXmlEntry, *61*
biodb::CsvFileConn, *76*, *95*
biodb::SqliteConn, *78*, *98*
BiodbConfig, *4*, 5, *70*
BiodbConn, 9, *34*, *45*, *61*, *84*, *106*
BiodbConnBase, *27*, 28, *35, 36*
BiodbCsvEntry, 34, *77*, *97*
BiodbDbInfo, *34*, 35, *38*
BiodbDbsInfo, *4*, *35, 36*, 36, *70*
BiodbEntry, *35*, 38, *61–63*, *73, 74*
BiodbEntryField, 46, *56*
BiodbEntryFields, *4*, *45, 46*, *53*, 53, *70*
BiodbFactory, *4*, *27*, *45*, 57, *70*
BiodbHtmlEntry, 61
BiodbJsonEntry, 62
BiodbListEntry, 63, *79*, *99*
BiodbMain, *4*, *9*, *36*, *38*, *53*, *56*, *61*, 63, *100*
BiodbSdfEntry, 71
BiodbTestMsgAck, 72
BiodbTxtEntry, *71*, 73
BiodbXmlEntry, *62*, 74

checkDeprecatedCacheFolders, 75
closeMatchPpm, 75
CompCsvFileConn, 76, *84*
CompCsvFileEntry, 77
CompSqliteConn, 78, *106*
CompSqliteEntry, 79
connNameToClassPrefix, 80
createBiodbTestInstance, 80
CsvFileConn, *76*, 81, *96*

df2str, 84

error, 85
error0, 85

FileTemplate, 86

getConnClassName, 87
getConnTypes, 88
getDefaultCacheDir, 88
getEntryClassName, 89
getEntryTypes, 89
getLogger, 90

listTestRefEntries, 90
loadFileContents, 91
logDebug, 91
logDebug0, 92
logInfo, 92
logInfo0, 93
logTrace, 93
logTrace0, 94
lst2str, 94

MassCsvFileConn, *84*, 95
MassCsvFileEntry, 97
MassSqliteConn, 98, *106*
MassSqliteEntry, 99

newInst, 100

Progress, 100

Range, 101
runGenericTests, 104

SqliteConn, *78*, *98*, 105

testContext, 106
TestRefEntries, 107
testThat, 109

warn, 110
warn0, 110