Package: mongo (via r-universe)

September 26, 2024

Type Package

Title Higher level interface to Mongo database

Version 0.9-5

Date 2024/05/25

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), methods, futile.logger

Imports jsonlite, mongolite

Suggests rlang, withr, knitr, rmarkdown, tidyr, CPTtools, bookdown, devtools, testthat (>= 3.0.0)

Description This is a wrapper for the jsonlite and mongolite packages which offers both an R6 object for managing the connection as well as some mechanisms for saving and restoring S4 objects to a Mongo database.

Collate as.json.R MongoDB.R jqmongo.R FakeMongo.R

URL https://github.com/ralmond/mongo

Encoding UTF-8

LazyData true

VignetteBuilder knitr

RoxygenNote 7.3.1

Config/testthat/edition 3

Support c('Bill & Melinda Gates Foundation grant ``Games as Learning/Assessment: Stealth Assessment" (#0PP1035331, Val Shute, PI)', 'National Science Foundation grant ``DIP: Game-based Assessment and Support of STEM-related Competencies" (#1628937, Val Shute, PI)', 'National Scient Foundation grant ``Mathematical Learning via Architectual Design and Modeling Using E-Rebuild." (\#1720533, Fengfeng Ke, PI)', 'Institute of Educational Statistics Grant: ``Exploring adaptive cognitive and

affective learning support for next-generation STEM learning games." (#R305A170376-20, Val Shute and Russell Almond, PIs')

Roxygen list(markdown = TRUE)

Repository https://ralmond.r-universe.dev

RemoteUrl https://github.com/ralmond/mongo

RemoteRef HEAD

RemoteSha 12736deb3b9d5d4ec90d58a77e7fd5915173a7e4

Contents

`mongo-package` *Higher level interface to Mongo database*

Description

This is a wrapper for the jsonlite and mongolite packages which offers both an R6 object for managing the connection as well as some mechanisms for saving and restoring S4 objects to a Mongo database.

Details

The DESCRIPTION file: This package was not yet installed at build time.

Index: This package was not yet installed at build time.

This package provides extensions to the ‘mongolite‘ and ‘jsonlite‘ package, specifically for saving and restoring S4 objects as JSON documents in a mongo database.

MongoDB class and methods

Using a `mongo` object in an R6 reference class presents a number of problems. In particular, there is potential race condition where the prototype object is built during load time, and the will try to make a connection to the database before the appropriate code is loaded. The `MongoDB` fixes this problem. It is a wrapper for the `mongo` object which is created when first used (by calling the `$db()` method).

The "mongo" package offers a number of generic functions which wrap the corresponding method of the `mongo` object: `mdbAggregate`, `mdbCount`, `mdbDisconnect`, `mdbDistinct`, `mdbDrop`, `mdbExport`, `mdbFind`, `mdbImport`, `mdbIndex`, `mdbInfo`, `mdbInsert`, `mdbIterate`, `mdbMapreduce`, `mdbRemove`, `mdbRename`, `mdbReplace`, `mdbRun`, and `mdbUpdate`. It also adds `showCollections` and `showDatabases` which operate like the corresponding Mongo shell commands. All of these functions are S4 generics with methods for both the `mongo::MongoDB` and `mongolite::mongo` objects.

Syntactic Sugar for Mongo Queries

The function `buildJQuery` is a helper function which creates JSON documents from R lists. For example, `buildJQuery(name="Fred",timestamp=list(gte=Sys.time()))` evaluates to `{ "name":"Fred", "timestamp":{ "$gte":[{"$date":1684370555315}] } }` The function `buildJQterm` is a helper, and `makeDBuri` is used for setting the ‘uri‘ (or ‘url‘) field for the `MongoDB` class.

JSON representation for S4 object

Saving an S4 object as a JSON document is a complex process. One approach is to use the functions `jsonlite::serializeJSON` and `jsonlite::unserializeJSON` methods. These will faithfully reproduce the object, but using the serialized object outside of R (in particular, in a Mongo collection or with the command line utility `jq`) is very difficult.

A second approach is to first turn the object into a list using `attributes(obj)`, and then using `jsonlite::toJSON` and `jsonlite::fromJSON` to do the conversion. One big problem with this approach is that these functions always turn R object into JSON arrays, even if the value is a scalar. Also, the types of the fields may not be the same after saving and restoring the object.

To fix the scalar/vector problem, the function `unbox` function marks a value as a scalar. The function `unboxer` is improved version which descends recursively through a complex list structure. The function `ununboxer` undoes the marking (mainly needed for testing). The functions `unparseData` and `parseSimpleData` provide some support for more complex structures.

The `as.json` function attempts to implement the second approach in a way that is transparent to the end user, but requires some effort on the part of the package designer. In particular, most S4 classes will require a custom `as.jlist` method which does appropriate transformations on the fields of the object (now elements of the list). The vignette "JSON for S4 objects" provides an example.

The function `parse.json` goes in the opposite direction. The JSON is converted into a list and then passed along to the builder function helper `buildObject`. The builder calls `parse.jlist` to reverse the `as.jlist` processing. The default builder then passes the list to the `new` function to create the new object. This approach should work well for most S4 objects, but S3 objects may need a custom constructor. In this case, the `buildObject` default builder needs to be replaced with custom code.

The vignette "JSON for S4 Objects" provides an extended example. The example `Event` and its associated `as.jlist` and `parse.jlist` methods are found in the file `system.file("examples","Event.R",package="mongo` This file can be loaded (used in some of the documentation examples) using the function `load_example` function.

Saving S4 objects in a database

Once the `as.json` and `parse.json` methods are built, saving the S4 object in a Mongo collection is straightforward; restoring the objects is also straightforward but requires the use of `mdbIterate` instead of `mdbFind` (the latter returns a data frame not a list).

The functions `saveRec`, `getOneRec`, and `getManyRecs` combine the calls to provide a straightforward mechanism for saving and restorting S4 objects from a database. S3 object may require a custom builder function (instead of `buildObject`), which can be passed as an optional argument to `getOneRec` or `getManyRecs`.

Mongo databases use the special field "_id" to provide a unique identifier for the collection. The class `MongoRec` is a simple class which provides that field (and can be used in the `contains` argument in the `setClass` method). The generic function `m_id` gets or sets the "_id" field. For objects that have not yet been added to the database, the value `NA_character_` should be used.

The `saveRec` method modified its behavior based on the value of the value of `m_id(obj)`. If this is missing, `saveRec` adds a new document to the collection, if it is present, then `saveRec` replaces the item in the collectin.

A mock MongoDB class

In designing a test suite for a class, it is often useful to 'mock' functions which rely on external resources with ones which have more predictable result. In particular, the `fake_mongo` is a drop in replacement for the `MongoDB` function which returns the results of various queries from a queue instead of a database.

The `iterator` class is a simple queue implementation. It is an R6 class with a collection of elements and a pointer to the next item in the collection. The `$hasNext()` method is a logical method that checks for more elements and `$nextElement()` which returns the next element (and updates the pointer). The `$reset()` method moves the pointer back to the beginning with and optional argument which replaces the element collection. It also has `$one()` and `$batch(n)` methods so it can mock the output of `mdbIterate`.

The `fake_mongo` class has queues corresponding to the functions which return messages other than status messages.

Method	Queue Name
`mdbAggregate`	"aggregate"
`mdbCount`	"count"
`mdbDistinct`	"distinct"
`mdbFind`	"find"
`mdbIterate`	"iterate"
`mdbMapreduce`	"mapreduce"
`mdbRun`	"run"
`showCollections`	"collections"
`showDatabases`	"databases"

The `$que(which)` method returns the `iterator` implementing the queue. The `$resetQue(which)` method resets a queue (with an optional arugment allowing to set a new collection of elements) and `$resetAll()` resets all queues.

Author(s)

Russell Almond

Maintainer: Russell Almond <ralmond@fsu.edu>

References

Mongolite User Manual: https://jeroen.github.io/mongolite/

Mongo DB command reference (make sure you look at the version corresponding to the mongo database used by your system). https://www.mongodb.com/docs/manual/reference/command/

See Also

`vignette("json-aaquickstart",package="jsonlite")` `mongolite`, `Proc4` (https://ralmond.r-universe.dev/ralmond/Proc4)

Examples

```
  ## Not run:
    vingette("JSON for S4 Objects")

## End(Not run)
```

---

as.json *Converts S4 objects to JSON representation.*

Description

These methods extend the toJSON function providing an extensible protocol for serializing S4 objects. The function as.json turns the object into a string containing a JSON document by first calling as.jlist to convert the object into a list and then calling toJSON to do the work.

Usage

```
as.json(
  x,
  serialize = TRUE,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "list"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  null = c("list", "null"),
  na = c("null", "string")
)

as.jlist(obj, ml, serialize = TRUE)

## S4 method for signature 'ANY'
as.json(
  x,
  serialize = TRUE,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "list"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  null = c("list", "null"),
  na = c("null", "string")
)
```

```
## S4 method for signature 'MongoRec'
as.json(
  x,
  serialize = TRUE,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "list"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  null = c("list", "null"),
  na = c("null", "string")
)

## S4 method for signature 'ANY,list'
as.jlist(obj, ml, serialize = TRUE)

## S4 method for signature 'MongoRec,list'
as.jlist(obj, ml, serialize = TRUE)
```

Arguments

x	An (S4) object to be serialized.
serialize	logical – Preserve all R information at the expense of legibility. Passed to `jsonlite::toJSON()`
dataframe	("rows", "columns", "values") – Order for data frames. Passed to `jsonlite::toJSON()`
matrix	("rowmajor" "columnmajor") – Order for matrix elements. Passed to `jsonlite::toJSON()`
Date	("ISO8601" "epoch") – Passed to `jsonlite::toJSON()`
POSIXt	("string" "ISO8601" "epoch" "mongo") – Date/time format. Passed to `jsonlite::toJSON()`
factor	("string" "list") – Treatment of factor variables. Passed to `jsonlite::toJSON()`
complex	("string" "list") – Representation for complex numbers. Passed to `jsonlite::toJSON()`
raw	("base64" "hex" "mongo" "int" "js") – Treatment of raw data. Passed to `jsonlite::toJSON()`
null	("list" "null") – Treatment of null fields. Passed to `jsonlite::toJSON()`
na	("null" "string") – Representation for NA's. Passed to `jsonlite::toJSON()`
obj	The object being serialized
ml	A list of fields of the object; usually `attributes(obj)`.

Details

The existing `toJSON` does not support S4 objects, and the `serializeJSON` provides too much detail; so while it is good for saving and restoring R objects, it is not good for sharing data between programs. The function `as.json` and `as.jlist` are S4 generics, so they can be easily extended to other classes.

The default method for `as.json` is essentially `toJSON( as.jlist(x, attributes(x)))`. The function `attributes(x)` turns the fields of the object into a list, and then the appropriate method for `as.jlist` further processes those objects. For example, it can set the `"_id"` field used by the Mongo DB as a unique identifier (or other derived fields) to `NULL`.

Another important step is to call `unboxer` on fields which should not be stored as vectors. The function `toJSON` by default wraps all R objects in '[]' (after all, they are all vectors), but that is probably not useful if the field is to be used as an index. Wrapping the field in `unboxer()`, i.e., using `ml$field <- unboxer(ml$field)`, suppresses the brackets. The function `unboxer()` in this package is an extension of the `jsonlite::unbox` function, which does not properly unbox POSIXt objects.

Finally, for a field that can contain arbitrary R objects, the function `unparseData` coverts the data into a JSON string which will completely recover the data. The `serialize` argument is passed to this function. If true, then `serializeJSON` is used which produces safe, but not particularly human editable JSON. If false, a simpler method is employed which produes more human readable code. This with should work for simpler data types, but does not support objects, and may fail with complex lists.

Value

The function `as.json` returns a unicode string with a serialized version of the object.

The function `as.jlist` returns a list of the fields of the object which need to be serialized (usually through a call to `toJSON`.

Methods (by class)

- `as.json(MongoRec)`: The `as.json` for `\linkS4class{MongoRec}` objects defaults to using "mongo" format for the `POSIXt` and `raw` options.
- `as.jlist(obj = ANY, ml = list)`: This is the default method, it simply returns the list of slots `ml`. This also does not contain a call to `callNextMethod`, so it will serve as the termination point for an inheritance chain.
- `as.jlist(obj = MongoRec, ml = list)`: This method actually removes the Mongo id (`_id`) as generally, that is not pass as part of an update query.

Author(s)

Russell Almond

See Also

In this package: `buildObject`, `saveRec`, `parseData`, `parseSimpleData`

In the jsonlite package: `toJSON`, `serializeJSON`, `jsonlite::unbox`

Examples

```
## Not run:
vingette("JSON for S4 Objects")

## End(Not run)
```

`buildJQterm` *Build a single query function.*

Description

Build a single query function.

Usage

```
buildJQterm(name, value)
```

Arguments

`name`	character name of the referenced field
`value`	vector named collection of possible values.

Details

This is mostly an internal function, but may be of some use.

Value

character scalar giving JSON expression

Examples

```
buildJQterm("uid","Fred")
buildJQterm("uid",c("Phred","Fred"))
buildJQterm("time",Sys.time())
buildJQterm("num",1:4)
buildJQterm("num",c(gt=7))
buildJQterm("num",c(lt=7))
buildJQterm("num",c(gte=7))
buildJQterm("num",c(lte=7))
buildJQterm("num",c(ne=7))
buildJQterm("num",c(eq=7))
buildJQterm("num",c(gt=2,lt=7))
buildJQterm("count",c(nin=1,2:4))
buildJQterm("count",c("in"=1,2:4))
buildJQterm("count",c(ne=1,ne=5))
```

`buildJQuery` *Transforms a query into JQuery JSON.*

Description

This function takes a query which is expressed in the argument list and transforms it into a JSON query document which can be used with the Mongo Database. The function `buildJQterm` is a helper function which builds up a single term of the query.

Usage

```
buildJQuery(..., rawfields = character())
```

Arguments

`...`	This should be a named list of arguments. The values should be the desired query value, or a more complex expression (see details).
`rawfields`	These arguments are passed as character vectors directly into the query document without processing.

Details

A typical query to a Mongo database collection is done with a JSON object which has a number of bits that look like "*field*:*value*", where *field* names a field in the document, and *value* is a value to be matched. A record matches the query if all of the fields specified in the query match the corresponding fields in the record.

Note that *value* could be a special expression which gives specifies a more complex expression allowing for ranges of values. In particular, the Mongo query language supports the following operators: `"$eq"`, `"$ne"`,`"$gt"`, `"$lt"`, `"$gte"`, `"$lte"`. These can be specified using a value of the form `c(<op>=<value>)`, where *op* is one of the mongo operators, without the leading '$'. Multiple op–value pairs can be specified; for example, `count=c(gt=3,lt=6)`. If no op is specified, then `"$eq"` is assumed. Additionally, the `"$oid"` operator can be used to specify that a value should be treated as a Mongo record identifier.

The `"$in"` and `"$nin"` are also ops, but the corrsponding value is a vector. They test if the record is in or not in the specified value. If the value is vector valued, and no operator is specified it defaults to `"$in"`.

The function `buildJQuery` processes each of its arguments, adding them onto the query document. The `rawfields` argument adds the fields onto the document without further processing. It is useful for control arugments like `"$limit"` and `"$sort"`.

Value

The function `buildJQuery` returns a unicode string which contains the JSON query document.

Author(s)

Russell Almond

References

The MongoDB 4.0 Manual: https://docs.mongodb.com/manual/

See Also

as.json, mdbFind, getOneRec, getManyRecs mongo

Examples

```
buildJQuery(app="default",uid="Phred")
buildJQuery("_id"=c(oid="123456789"))
buildJQuery(name="George",count=c(gt=3,lt=5))
buildJQuery(name="George",count=c(gt=3,lt=5),
            rawfields=c('"$limit":1','"$sort":{timestamp:-1}'))

## Queries on IDs need special handling
buildJQuery("_id"=c(oid="123456789abcdef"))
```

---

codeClass *Adds/removes package information to class descriptions*

Description

If the class has a "package" attribute, then changes the descriptor to a form "package::classname", e.g., the MongoRec class, which lives in the "mongo" package becomes mongo::MongoRec. The function decodeClass() reverses this.

Usage

```
codeClass(class)

decodeClass(class)
```

Arguments

class	character Class identifiers. For codeClass() these might have the "package" attribute set. For decodeClass() the package attribute is represented by the prefix "package::".

Value

The function codeClass() returns a character vector with "package" attributes changed to "package::" prefixes. The function decodeClass() returns a character vector with "package::" prefixes removed and "package" attributes set.

Note

The codeClass() function applies unboxer() to mark single class names as singletons. The decodeClass() function applies ununboxer() to remove the mark if needed. Also, if class is a list (happens if it was not quoted with unboxer() when saved to JSON), decodeClass() will try to coerce it into a character vector.

Examples

```
codeClass(class(MongoRec()))
codeClass(class(matrix(1:4,2,2)))
decodeClass(codeClass(class(MongoRec())))
decodeClass(codeClass(class(matrix(1:4,2,2))))
```

---

fake_mongo-class *A simulated* MongoDB *object for testing*

Description

This class simulates the behavior of a mongo collection providing a set of scripted responses to queries. In particular, mdbAggregate(), mdbCount(), mdbDistinct(), mdbFind(), mdbIterate(), mdbMapreduce(), mdbRun(), showCollections() and showDatabases() methods are overridden to return prespecified results in order. Usually, no connection is made to an actual database, so this can be used to run tests in environments where it is unknown whether or not an appropriate mongo database is available.

Usage

```
fake_mongo(
  collection = "test",
  db = "test",
  url = "mongodb://localhost",
  verbose = FALSE,
  options = mongolite::ssl_options(),
  noMongo = TRUE,
  logging = TRUE,
  aggregate = list(),
  count = list(),
  distinct = list(),
  find = list(),
  iterate = list(),
  mapreduce = list(),
  run = list(),
  databases = list(),
  collections = list()
)
```

```
## S4 method for signature 'fake_mongo'
mdbAvailable(db)

## S4 method for signature 'fake_mongo'
mdbAggregate(
  db,
  pipeline = "{}",
  options = "{\"allowDiskUse\":true}",
  handler = NULL,
  pagesize = 1000,
  iterate = FALSE
)

## S4 method for signature 'fake_mongo'
mdbCount(db, query = "{}")

## S4 method for signature 'fake_mongo'
mdbDisconnect(db)

## S4 method for signature 'fake_mongo'
mdbDistinct(db, key, query = "{}")

## S4 method for signature 'fake_mongo'
mdbDrop(db)

## S4 method for signature 'fake_mongo'
mdbExport(
  db,
  con = stdout(),
  bson = FALSE,
  query = "{}",
  fields = "{}",
  sort = "{\"_id\":1}"
)

## S4 method for signature 'fake_mongo'
mdbImport(db, con = stdout(), bson = FALSE)

## S4 method for signature 'fake_mongo'
mdbFind(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0,
  handler = NULL,
  pagesize = 1000
```

```
)

## S4 method for signature 'fake_mongo'
mdbIndex(db, add = NULL, remove = NULL)

## S4 method for signature 'fake_mongo'
mdbInfo(db)

## S4 method for signature 'fake_mongo'
mdbInsert(db, data, pagesize = 100, stop_on_error = TRUE, ...)

## S4 method for signature 'fake_mongo'
mdbIterate(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0
)

## S4 method for signature 'fake_mongo'
mdbMapreduce(db, map, reduce, query = "{}", sort = "{}", limit = 0)

## S4 method for signature 'fake_mongo'
mdbRemove(db, query = "{}", just_one = FALSE)

## S4 method for signature 'fake_mongo'
mdbRename(mdb, name, db = NULL)

## S4 method for signature 'fake_mongo'
mdbReplace(db, query, update = "{}", upsert = FALSE)

## S4 method for signature 'fake_mongo'
mdbUpsert(db, query, update = "{}", upsert = TRUE)

## S4 method for signature 'fake_mongo'
mdbRun(db, command = "{\"ping\":1}", simplify = TRUE)

## S4 method for signature 'fake_mongo'
mdbUpdate(
  db,
  query,
  update = "{\"$set\":{}}",
  filters = NULL,
  upsert = FALSE,
  multiple = FALSE
)
```

```
## S4 method for signature 'fake_mongo'
showDatabases(
  db = NULL,
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)

## S4 method for signature 'fake_mongo'
showCollections(
  db = NULL,
  dbname = "test",
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)
```

Arguments

collection	character – name of the referenced collection
db	character – name of the referenced database
url	character – URI for accessing the database.
verbose	logical – passed to \link[mongolite]{mongo}
options	ANY – SSL options passed to mongo call.
noMongo	logical – If true (default), no attempt is made to connect to the Mongo database.
logging	logical – If true (default), then calls to the database will be logged.
aggregate	list – simulated responses from mdbAggregate() queries.
count	list – simulated responses from mdbCount() queries.
distinct	list – simulated responses from mdbDistinct() queries.
find	list – simulated responses from mdbFind() queries.
iterate	list – simulated responses from mdbIterate() queries.
mapreduce	list – simulated responses from mdbMapreduce() queries.
run	list – simulated responses from mdbRun() queries.
databases	list – simulated responses from showDatabases() queries.
collections	list – simulated responses from showCollections() queries.
pipeline, handler, pagesize, query, key, fields, sort, skip, limit, map, reduce, command, simplify, uri, dbname, con, bson, add, remove, data, stop_on_error, just_one, mdb, name, update, upsert, filters, multiple, ...	– arguments to the generic functions which are ignored in the fake_mongo methods.

Details

Internally the `fake_mongo` class has a list of iterators named "aggregate", "count", "distinct", "find", "iterate", "mapreduce", "run", "databases", and "collections". The corresponding methods will return the next entry in the iterator (if it exists) or else will call the parent method to get the default return value (varies with generic function). Usually, no connection to a mongo database is made.

The Queue names are given in the following table.

Method	Queue Name
`mdbAggregate`	"aggregate"
`mdbCount`	"count"
`mdbDistinct`	"distinct"
`mdbFind`	"find"
`mdbIterate`	"iterate"
`mdbMapreduce`	"mapreduce"
`mdbRun`	"run"
`showCollections`	"collections"
`showDatabases`	"databases"

These names are used as `which` arguments to the `$que(which)` and `$resetQueue(which)` methods as well as for initializing the queue using the `fake_mongo` constructor.

If logging is turned on (either by setting `logging=TRUE` in the constructor, or by calling `$logging(TRUE)`, then each `mdbXXX` method will log the call to the `log` collection. The `$getLog()` and `$lastLog()` methods access the queue, and `$resetQueue()` resets it.

Value

An object of class `fake_mongo`

An object of type ‘fake_mongo

Functions

- `fake_mongo()`: Constructor

Fields

`queues` a named list of `\linkS4class{iterator}` objects which provide the simulated responses.

`log` a list of database calls made

`logp` logical If `TRUE` then method class will be logged.

Class-Based Methods

- `$initialize(...)` – See `fake_mongo` function for arguments.
- `$que(which)` – Returns an individual response queue as an `\linkS4class{iterator}` The `which` argument should be one of the names in the table in the Details session.

- `$resetQueue(which, newElements=NULL)` – Calls the `$reset()` method on `\linkS4class{iterator}` associated with operation `which`. If `newElements` is supplied, the elements or the iterator are replaced. Note that if `which="iterate"`, then the queue is an iterator which returns iterators. The `$reset()` method is called on all of the elements of the queue as well.
- `$resetAll()` – Resets all Queues.
- `$logging(newState)` Checks or sets the logging state. If the argument is supplied, this sets the state.
- `logCall(call)` – Logs a database CRUD operation. The `call` argument is a named list. The first element, named `op` is the database operation (the name of the call minus the `mdb` prefix). The remaining arguments are the values of the arguments in the CRUD call.
- getLog(newestFirst=TRUE) – Fetches the entire log. Log is stored with the newest call first (reverse chronological order), so this is the default order.
- `$lastLog()` – Returns the most recently added element in the log.
- `$resetLog()` – Clears the call log.

Methods

This class overrides all of the normal CRUD (`mdbXXX`) methods.

For all methods, the internal `$logCall()` method is called giving the details of the call.

For the methods which correspond to a queue, the next element in the corresponding queue will be returned.

This allows faking the database connection to test functions which interact with the mongo database.

Examples

```
showClass("fake_mongo")
```

`getOneRec` *Fetches Messages from a Mongo databas*

Description

This function fetches `MongoRec` objects from a `mongo` database. The message parser is passed as an argument, allowing it to fetch other kinds of objects than P4Messages. The function `getManyRecs` retrieves all matching objects and the function `getOneRec` retrieves the first matching object.

Usage

```
getOneRec(
  col,
  jquery = "{}",
  builder = buildObject,
  sort = buildJQuery(timestamp = -1)
)
```

```
getManyRecs(
  col,
  jquery,
  builder = buildObject,
  sort = buildJQuery(timestamp = -1),
  skip = 0,
  limit = 0
)
```

Arguments

col	(or MongoDB mongo) A reference to a Mongo collection.
jquery	A string providing a Mongo JQuery to select the appropriate records. See buildJQuery.
builder	A function which will take the list of fields returned from the database and build an appropriate R object. See buildObject.
sort	A named numeric vector giving sorting instructions. The names should correspond to fields of the objects, and the values should be positive or negative one for increasing or decreasing order. Use the value NULL to leave the results unsorted.
skip	integer This many records should be skipped before returning records
limit	A numeric scalar giving the maximum number of objects to retrieve. If Inf, then all objects matching the query will be retrieved.

Details

This function assumes that a number of objects (usually, but not necessarily subclasses of MongoRec objects) have been stored in a Mongo database. The col argument is the MongoDB object in which they are stored. These functions retrieve the selected objects.

The first argument should be a string containing a JSON query document. Normally, thes are constructed through a call to buildJQuery.

The query is used to create an iterator over JSON documents stored in the database. At each round, the iterator extracts the JSON document as a (nested) list structure. This is passed to the builder function to build an object of the specified type. See the buildObject function for an example builder.

The sorting argument controls the way the returned list of objects is sorted. This should be a numeric vector with names giving the field for sorting. The default values c("timestamp"=1) and c("timestamp"=-1) sort the records in ascending and descending order respectively. In particular, the default value for getOneRec means that the most recent value will be returned. The defaults assume that "timestamp" is a field of the stored object. To suppress sorting of outputs, use NULL as the argument to sort.

Value

The function getOneRec returns an object whose type is determined by the output of the builder function. The default \link{buildObject} method uses the class field of the record is used to select the object type. (It assumes a \link{parse.jlist} method is available for that object type.)

The function getManyRecs returns a list of object whose type is determined by the output of the builder function.

Author(s)

Russell Almond

References

The MongoDB Manual: https://docs.mongodb.com/manual/

See Also

saveRec, buildObject, getOneRec, getManyRecs mongo

Examples

```
## Not run:
## Requires Mongo test database to be set up.
load_Events()

m1 <- new("Event", uid="James Goodfellow",mess="Task Done",processed=FALSE,
           timestamp=Sys.time(),
           data=list("Selection"="B"))
m2 <- new("Event", uid="James Goodfellow", mess="New Obs", processed=FALSE,
           timestamp=Sys.time(),
           data=list("isCorrect"=TRUE,"Selection"="B"))
m3 <- new("Event", uid="Fred",mess="New Stats",
           timestamp=Sys.time(),
           data=list("score"=1,"theta"=0.12345,"noitems"=1))

EventDB <- MongoDB(Event,noMongo=!interactive())

Assign these back to themselves to capture the mongo ID
m1 <- saveRec(EventDB,m1)
m2 <- saveRec(EventDB,m2)
m3 <- saveRec(EventDB,m3)

m1@data$time <- list(tim=25.4,units="secs")
m1 <- saveRec(EventDB,m1)

## Note use of oid keyword to fetch object by Mongo ID.
m1a <- getOneRec(EventDB,buildJQuery("_id"=c(oid=m1@"_id")))
m123 <- getManyRecs(EventDB,buildJQuery(uid="Fred"))
m23 <- getManyRecs(EventDB,buildJQuery(uid="Fred",sender=c("EI","EA")))
m321 <- getManyRecs(EventDB,buildJQuery(uid="Fred",timestamp=c(lte=Sys.time())),
            sort=c(timestamp=-1))
getManyRecs(EventDB,buildJQuery(uid="Fred",
                          timestamp=c(gte=Sys.time()-as.difftime(1,units="hours"))))
```

```
## End(Not run)
```

`iterator-class` *An object which iterates over a collection*

Description

An `iterator` loops through a collection using the `$hasNext()` and `$nextElement()` methods. This class also supports the `$one()` and `$batch()` methods to mimic the iterator returned by the `\link[mongolite]{mongo}()$iterate()` method.

Usage

```
iterator(elements = list())
```

Arguments

`elements`	A list of elements for the iterator to return.

Value

An object of class iterator.

The newly created iterator.

Fields

`elements` list – The objects to return

`position` integer – a pointer to the last returned object

Class-based Methods

- `$initialize(elements=list(),...)` Constructor
- `$hasNext()` – Logical value. Checks whether there are unseen elements in the collection. Position is not advanced.
- '$nextElement(warn=TRUE)`Returns the next item in the collection and advances the position. If no it`
- `$one()` Returns the next item in the collection. Designed to mimic the return from the `\link[mongolite]{mongo}$iterate()` function. The next object, or `NULL` (without a warning) if the collection is empty.
- `$batch(count)` Fetechs `count` elements as a list, advancing the position by the argument. Issues a warning if there are not `count` elements left in the collection. Advances the position by `count`.
- `$reset()` Resets the position back to the beginning. If an argument is supplied, it also replaces the `elements`.

Note

This is a utility class that serves two purposes. (1) It implements a result queue for the `linkS4class{fake_mongo}` class. (2) it mimics the iterator returned by the `mdbIterate()` generic function, and so can be used in the result queue for the `mdbIterate-fake_mongo` method.

Unlike the internal iterator class from the `mongolite`, this one has a `$hasNext()` method which is part of the general iterator recipe. The `$one()` and `$batch()` methods should be compatible with the internal `mongolite` iterator, can so it can be used as drop in replacement.

See Also

`mdbIterate()`, `\linkS4class{fake_mongo}`

Examples

```
iter <- iterator(as.list(1:5))
while (iter$hasNext())
  print(iter$nextElement())
```

`jlist` *List representation of a document.*

Description

A `namedList` which corresponds to a Mongo document; this is not an official type, but rather a particular use of use of the primitive `namedList` type. The field names are given by the names and the values are the list values. If any of the elements is a list, then it is a sub-document. The `jsonlite` package provides a `toJSON` and `fromJSON` method for converting between jlists and JSON character objects.

Details

Note that R makes no distinction between scalars and vectors of length 1; however, JSON does. For example, `'{"scalar":0, "vector":[0]}'`. The `jsonlite` package provides a tool `\link[jsonlite]{unbox}` which marks the element as a scalar. The function `\link{unboxer}` will do this recursively over a jlist object.

The distinction between vectors and scalars is unimportant if the goal is simply to save and restore the object, but if the goal is to build an index over the field (`\link{mdbIndex}`), then scalars are easier to work with than vectors. To covert an S4 object to a class, the solution is to write a method for the `\link{as.jlist}` method which makes appropriate transformations of the fields (and a `\link{parse.jlist}` method to reverse the process).

See Also

`\link{as.jlist}`, `\link{buildObject}`, `\link[jsonlite]{toJSON}`, `\link[jsonlite]{fromJSON}`, `\link{mdbIterate}`, `\link[jsonlite]{unbox}`, `\link{unboxer}`

JSONDB-class *Class which supports the mdbCRUD methods.*

Description

Class which supports the mdbCRUD methods.

Details

The CRUD (Create, Read, Update and Delete) are the basic set of operators for manipulating a database. The `mongo` package defines a number of operators (mostly named `mdbXXX`) which calls the corresponding CRUD operations. The `JSONDB` class is intended for any class that supports these operations.

The following operations are supported:

- `mdbAvailable()` – Returns logical value. If false, CRUD operations will basically be no-ops.
- `mdbAggregate()` – Runs an aggregation pipeline
- `mdbCount()` – Counts records matching query
- `mdbDisconnect()` – Drops connection to database (will be reconnected on next operation).
- `mdbDistinct()` – Lists unique values of a field.
- `mdbDrop()` – Drops the collection from the database. This is a fast way to clear a database.
- `mdbImport()`,`mdbExport()` – Imports/Exports documents into/from a collection from a file (or connection).
- `mdbFind()` – Finds documents matching query and returns result as a `data.frame`.
- `mdbIndex()` – Adds or removes an index from a collection.
- `mdbInfo()` – Returns info about a collection.
- `mdbInsert()` – Adds one or more documents into a collections. Works with both `data.frame` (one document per row) and JSON character vectors (one document per element).
- `mdbIterate()`/`mdbFindL()` – Finds documents matching query and returns these as an iterator/list.
- `mdbMapreduce()` – Executes a mapreduce operation using javascript map and reduce oerations.
- `mdbRemove()` – Removes matching documents from a collection.
- `mdbRename()` – Renames a collection.
- `mdbReplace()`/`mdbUpsert()` – Replaces a document in a collection.
- `mdbRun()` – Runs a Mongo command.
- `mdbUpdate()` – Modifies records in a database.
- `showDatabases()` – Lists available databases.
- `showCollections()` – Lists collections in a database.

load_example — *Load example Event class*

Description

This function loads the example "Event" class; needed for examples.

Usage

```
load_example()
```

Value

invisible details about package

Examples

```
load_example()
fred1  ## sample data item.
## The source file
system.file("examples","Event.R",package="mongo")
```

makeDBuri — *Creates the URI needed to connect to a mongo database.*

Description

This function formats the universal record indicator (URI) for connecting to a Mongo database. It is mostly a utility function for formatting the string.

Usage

```
makeDBuri(
  username = "",
  password = "",
  host = "localhost",
  port = "",
  protocol = "mongodb"
)
```

Arguments

`username`	The name of the database user (login credential), or an empty string if no username is required.
`password`	The name of the database password (login credential), or an empty string if no password is required.
`host`	The name or IP address of the system hosting the database.
`port`	The port to be used for connections. Note that the port for a default configuration of mongo is 27018. This can be left blank to use the default port.
`protocol`	A character scalar giving the protocol to use when connecting, e.g., "mongodb".

Value

A character string giving the database URI which can be passed to the `mongo` function to create a database collection handle.

Note that the password is stored in clear text, so appropriate care should be taken with the result of this function.

Author(s)

Russell Almond

See Also

`MongoDB`, `mongo`

This is an input argument to a number of other classes which use mongo connections.

Examples

```
makeDBuri()
makeDBuri(user="admin",password="secret")
makeDBuri(user="admin")
makeDBuri(host="example.com",port=12345)
```

---

mdbAggregate *Execute Aggregration Pipeline*

Description

Execute Aggregration Pipeline

Usage

```
mdbAggregate(
  db,
  pipeline = "{}",
  options = "{\"allowDiskUse\":true}",
  handler = NULL,
  pagesize = 1000,
  iterate = FALSE
)

## S4 method for signature 'MongoDB'
mdbAggregate(
  db,
  pipeline = "{}",
  options = "{\"allowDiskUse\":true}",
  handler = NULL,
  pagesize = 1000,
  iterate = FALSE
)

## S4 method for signature 'mongo'
mdbAggregate(
  db,
  pipeline = "{}",
  options = "{\"allowDiskUse\":true}",
  handler = NULL,
  pagesize = 1000,
  iterate = FALSE
)
```

Arguments

db	MongoDB or mongo – The database collection handle.
pipeline	character – a json object describing the pipeline.
options	character – a json object giving options to the pipeline. (This is missing from the mongolite documentation, so see Mongo documentation).
handler	– undocumented.
pagesize	integer – Size of pages
iterate	logical – If TRUE return an iterator (see \link{mdbIterate}), if false a data.frame.

Details

Execute a pipeline using the Mongo aggregation framework. Set iterate = TRUE to return an iterator instead of a data frame.

Value

Data frame or iterator with query results.

See Also

[mongo] https://www.mongodb.com/docs/manual/reference/command/aggregate/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
stats <- mdbAggregate(irisdb,
  paste('[{"$group":{"_id":"$Species", "count": {"$sum":1},',
                    '"average_Petal_Length": {"$avg":"$Petal_Length"}',
                    '}}]'),
  options = '{"allowDiskUse":true}'
)
if (!is.null(stats)) {
  names(stats) <- c("Species", "Count", "Petal Length")
}
print(stats)
```

---

mdbAvailable *Is the collection available for writing.*

Description

Returns FALSE if the connection to the database is not available, so the CRUD operations (mdbCRUD). will not be executed.

Usage

```
mdbAvailable(db)

## S4 method for signature 'MongoDB'
mdbAvailable(db)

## S4 method for signature 'mongo'
mdbAvailable(db)
```

Arguments

db	(or MongoDB mongo) – Reference to collection

Value

logical value. If false, there is no active connection and CRUD operations will be no-ops.

Note

When using the mongolite::mongo collection reference operations are not skipped.

mdbCount *Counts the number of records matching Query.*

Description

Counts the number of records matching Query.

Usage

```
mdbCount(db, query = "{}")

## S4 method for signature 'MongoDB'
mdbCount(db, query = "{}")

## S4 method for signature 'mongo'
mdbCount(db, query = "{}")
```

Arguments

db	MongoDB or mongo – Reference to the collection
query	character – JSON expression giving the query. See `\link{buildJQuery}`.

Details

The `query` argument is a partial match for the records (in JSON format) which is essentially a partial match for the object.

Value

integer The number of records found (or `NA` if `noMongo = TRUE`)

See Also

`\link[mongolite]{mongo}`, `\link{buildJQuery}` https://www.mongodb.com/docs/manual/reference/command/count/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbCount(irisdb)
mdbCount(irisdb,'{"Species":"setosa"}')
mdbCount(irisdb,buildJQuery(Sepal.Width=c(lt=3),Petal.Width=c(gt=.3,lt=1.8)))
```

`mdbDisconnect` *Disconnects connection to database.*

Description

Disconnects connection to database.

Usage

```
mdbDisconnect(db, gc = TRUE)

## S4 method for signature 'MongoDB'
mdbDisconnect(db, gc = TRUE)

## S4 method for signature 'mongo'
mdbDisconnect(db, gc = TRUE)
```

Arguments

db	MongoDB or mongo – The database connection to drop.
gc	logical – Should the garbage collection be run.

Details

While this closes the connection, the `MongoDB` object retains the information needed to re-open it. It will be reopened on the next call.

Value

status message

See Also

[mongo]

Examples

```
## Setting noMongo=TRUE, so we don't actually run this.
testDB <- MongoDB("test", noMongo=!interactive())
mdbDisconnect(testDB)
```

mdbDistinct *Find the distinct values of a particular field*

Description

Find the distinct values of a particular field

Usage

```
mdbDistinct(db, key, query = "{}")

## S4 method for signature 'MongoDB'
mdbDistinct(db, key, query = "{}")

## S4 method for signature 'mongo'
mdbDistinct(db, key, query = "{}")
```

Arguments

db	(or MongoDB mongo) – Reference to database collection
key	character – field to extract
query	character – JSON expression indicating subcollection. See \link{buildJQuery}.

Details

Finds the unique values of the field specified by key. If query is supplied, then search is restricted to records satisfying query.

Value

list of values

See Also

[mongo] https://www.mongodb.com/docs/manual/reference/command/distinct/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbDistinct(irisdb,"Species")
```

mdbDrop *Drops the database collection*

Description

Drops the database collection

Usage

```
mdbDrop(db)

## S4 method for signature 'MongoDB'
mdbDrop(db)

## S4 method for signature 'mongo'
mdbDrop(db)
```

Arguments

db	(or MongoDB mongo) – Reference to collection to drop

Details

Dropping the collection and then inserting values is an easy way to reset the collection contents, so is a common idiom.

Value

miniprint object giving status

See Also

[mongo] https://www.mongodb.com/docs/manual/reference/command/drop/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
```

mdbExport *Exports/imports data from external JSON or BSON file*

Description

Exports/imports data from external JSON or BSON file

Usage

```
mdbExport(
  db,
  con = stdout(),
  bson = FALSE,
  query = "{}",
  fields = "{}",
  sort = "{\"_id\":1}"
)

## S4 method for signature 'MongoDB'
mdbExport(
  db,
  con = stdout(),
  bson = FALSE,
  query = "{}",
  fields = "{}",
  sort = "{\"_id\":1}"
)

## S4 method for signature 'mongo'
mdbExport(
  db,
  con = stdout(),
  bson = FALSE,
  query = "{}",
  fields = "{}",
  sort = "{\"_id\":1}"
)

mdbImport(db, con, bson = FALSE)

## S4 method for signature 'MongoDB'
mdbImport(db, con, bson = FALSE)

## S4 method for signature 'mongo'
mdbImport(db, con, bson = FALSE)
```

Arguments

db	(or MongoDB mongo) – Database collection of focus
con	connetion – a file or other connection for import/export
bson	logical – If TRUE use BSON (binary JSON) if FALSE then JSON
query	character – JSON expression providing a query selecting records to export. See \link{buildJQuery}.
fields	character – JSON expression selecting fields of the objects to be exported. See \link{mdbFind} for details.
sort	character – JSON expression indicating field and direction for sorting exported records. See \link{mdbFind} for details.

Details

The export function dumps a collection to a file or other connection. This can be in either plain text (utf8) JSON format, or a binary BSON format (this is specific to Mongo). The import function reverses this process.

On export, the query, fields and sort fields can be used to control what is exported.

Value

miniprint object giving the status

See Also

[mongo] https://www.mongodb.com/docs/manual/reference/command/import/ https://www.mongodb.com/docs/manual/reference/command/export/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbCount(irisdb)
outfile <- tempfile(fileext="json")
mdbExport(irisdb,file(outfile),sort='{"Petal_Length":-1}')
mdbDrop(irisdb)
mdbCount(irisdb)
mdbImport(irisdb,file(outfile))
mdbCount(irisdb)
```

mdbFind *Finds records which match the query and returns as data frame*

Description

Finds records which match the query and returns as data frame

Usage

```
mdbFind(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0,
  handler = NULL,
  pagesize = 1000
)

## S4 method for signature 'MongoDB'
mdbFind(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0,
  handler = NULL,
  pagesize = 1000
)

## S4 method for signature 'mongo'
mdbFind(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0,
  handler = NULL,
  pagesize = 1000
)
```

Arguments

db	(or MongoDB mongo) – the database collection.

query	character – A query string in JSON format. See \link{buildJQuery}.
fields	character – A JSON expression describing which fields to extract from the selected records. The default returns all fields except for the internal Mongo id field (see \link{m_id}.
sort	character – A JSON field indicating how the record should be sorted.
skip	integer – The number of records to skip.
limit	integer – The maximum number of records to return.
handler	(or NULL function) – Undocumented.
pagesize	integer – Used for buffering

Details

The mdbFind function takes a collection of records and turns it into a data.frame with the columns representing the frame. To process the raw JSON stream, try \link{mdbIterate}

The query, fields and sort are all JSON expressions. Note that field names shuld be quoted inside these query strings (the quotes are optional in the Mongo shell, but not here). I recommend using single quotes for the outer expression and double quotes inside the JSON string.

Query:

The mongo query is a rather rich language. The simplest version restricts a field to a specific value. For example '{"Species":"virginica"}' would select only virginica irises. There are a number of different operators which can used to specify the query, for examples '{"Species":{$ne:"virginica"}}' and '{"Species":{$in:["setosa","versicolor"]}}' both select the other iris types.

The mongo operators are "$eq" – equals, "$gt" – greater than, "$gte" – greater than or equals, "$lt" – less than, "$lte" – less than or equals, "$ne" – not equal,"$nin" – not in (argument is a list ('[]'),"$in" – in (argument is a list) and "$regex" – a regular expression.

The function \link{buildJQuery} uses a more R-like syntax and converts them to JSON. This makes it easier to build a query inside of R.

fields:

The fields JSON expression should be a collection of fields with a true or false (or 0 or 1). Note that the "_id" field is automatically included unless explicitly excluded. For example: {"Petal.Length":1, "Petal.Width":1, "Species":1, "_id":0} will select the petal length and width field and species field. See the topic Projection in the Mongo manual for more details.

sort:

This is a short object which gives the name of the field to sort on and the direction (1 for ascending, -1 for descending). If more than one sort key is given, the first one is given the highest priority. The sort keys should be included in the field. For example {"Petal.Length":1} sort in ascending order according to petal length. See the sort function in the Mongo reference manual.

Value

data.frame giving query results

See Also

\link[mongolite]{mongo}, \link{buildJQuery} \link{getOneRec}, \link{getManyRecs}, \link{mdbIterate} https://www.mongodb.com/docs/manual/reference/operator/query/ https://www.mongodb.com/docs/manual/reference/command/find/ https://www.mongodb.com/docs/manual/reference/method/db.collection.find/#std-label-find-projection https://www.mongodb.com/docs/manual/reference/method/cursor.sort/#mongodb-method-cursor.sort

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbFind(irisdb,buildJQuery(Species="setosa"),
        fields = '{"Petal_Width":1, "Petal_Length":1}',
        sort = '{"Petal_Width":-1}', limit=10)
```

---

mdbIndex *Build/remove an index for the collection.*

Description

Build/remove an index for the collection.

Usage

```
mdbIndex(db, add = NULL, remove = NULL)

## S4 method for signature 'MongoDB'
mdbIndex(db, add = NULL, remove = NULL)

## S4 method for signature 'mongo'
mdbIndex(db, add = NULL, remove = NULL)
```

Arguments

db	(or MongoDB mongo) – Collection in question.
add	character – JSON object describing fields to index.
remove	character – Name of indexes to remove.

Details

If add is specified, then a new index is added. If remove then the index is removed. If neither is specified, then a data frame giving the existing indexes is returned.

The syntax of the add field is similar to the sort argument of \link{mdbFind}. The removefunction uses thename‘ from the returned data.frame.

If sorted queries are going to be frequent, then building indexes will improve performance.

Value

data frame desribing indexes.

See Also

[mongo] https://www.mongodb.com/docs/manual/reference/method/db.collection.createIndex/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbIndex(irisdb,add=buildJQuery("Petal_Length"=1))
mdbIndex(irisdb,add='{"Petal_Length":1,"Petal_Width":-1}')
indexes <- mdbIndex(irisdb)
print(indexes)
mdbIndex(irisdb,remove="Petal_Length_1")
```

---

mdbInfo *Get Information about the collection*

Description

Get Information about the collection

Usage

```
mdbInfo(db)

## S4 method for signature 'MongoDB'
mdbInfo(db)

## S4 method for signature 'mongo'
mdbInfo(db)
```

Arguments

db	(or MongoDB mongo) The collection of interest.

Value

Object of class `miniprint` giving information about the collection.

See Also

[mongo]

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbInfo(irisdb)
```

---

mdbInsert *Insert a new record into a collection*

Description

Inserts one or more records. If the `data` argument is a `data.frame`, then each row becomes a new record.

Usage

```
mdbInsert(db, data, pagesize = 1000, stop_on_error = TRUE, ...)

## S4 method for signature 'MongoDB'
mdbInsert(db, data, pagesize = 1000, stop_on_error = TRUE, ...)

## S4 method for signature 'mongo'
mdbInsert(db, data, pagesize = 1000, stop_on_error = TRUE, ...)
```

Arguments

`db`	(or MongoDB mongo) – Collection into which new recrods will be inserted
`data`	(or data.frame named list character) – New data to be inserted.
`pagesize`	integer – size of data stores
`stop_on_error`	logical
`...`	– extra data

Details

Data frames:

Data frames are converted into mongo documents and then inserted. Each row is a document, and the fields in the document correspond to properties. This is perhaps the easiest way to use this function. `mdbInsert` save a data frame in a mongo collection and `\link{mdbFind}` retrieves it.

Character:

An alternative is to express the document to be stored as a JSON string. If the input is a character vector with each element being a complete JSON document, these will be added to the collection. The function `\link[jsonlite]{serializeJSON}` in the jsonlite package converts an R object to JSON in a way that will reproduce the object but is not particularly easy to find or index in the database. The function `\link{jsonlite}{toJSON}` produces a more readable version, but still

has issues (in partuclar, it does not distinguish between scalar and vector fields). The function \link{as.json} provides a mechanism for encoding S4 objects as JSON expressions.

The function \link{saveRec} provides a more object-oriented interface for saving a single S4 object.

List:

The source code for \link[mongolite]{mongo}()$insert() provides a mechanism for using lists, but does not describe what the lists elements should be.

Value

Object of class miniprint giving status information.

See Also

\link[mongolite]{mongo}, \link[jsonlite]{toJSON}, \link[jsonlite]{serializeJSON}, \link{as.json}, \link{saveRec} https://www.mongodb.com/docs/manual/reference/method/db.collection.insert/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
testdb <- MongoDB("test",noMongo=!interactive())
mdbDrop(testdb)
mdbInsert(testdb,'{"Student":"Fred", "Scores":[83, 87, 91, 79],
"Grade":"B"}')
```

---

mdbIterate *Returns documents as lists (jlists) from the database.*

Description

Returns documents as lists (jlists) from the database.

Usage

```
mdbIterate(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0
```

```
)

## S4 method for signature 'MongoDB'
mdbIterate(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0
)

## S4 method for signature 'mongo'
mdbIterate(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0
)

mdbFindL(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0
)

## S4 method for signature 'JSONDB'
mdbFindL(
  db,
  query = "{}",
  fields = "{\"_id\":0}",
  sort = "{}",
  skip = 0,
  limit = 0
)
```

Arguments

db	(or MongoDB mongo) – the database collection.
query	character – A query string in JSON format. See \link{buildJQuery} and mdbFind.
fields	character – A JSON expression describing which fields to extract from the selected records. The default returns all fields except for the internal Mongo id

	field (see \link{m_id}.
sort	character – A JSON field indicating how the record should be sorted.
skip	integer – The number of records to skip.
limit	integer – The maximum number of records to return.

Details

Unlike the \link{mdbFind} operation, which converts to the query output to a data frame, mdbIterate produces an iterator, which will cycle through the query results, which are returned as \link{jlist} objects.

Value

An iterator with the values.

Iterators

The iterator object returned from this function has two methods:

- $one() – Returns the next object, or NULL if there is none.
- $batch(n) – Returns the next n objects.

See Also

[mongo]

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
iter <- mdbIterate(irisdb,limit=10)
if (!is.null(iter)) {
iter$one()
iter$batch(3)
## Note extra parens.
while (!is.null((item <- iter$one()))) {
  print(sprintf("A %s iris with petal length %f",
                item$Species,item$Petal_Length))
}
}
```

mdbMapreduce *Applies a summary operation to a collection*

Description

Runs a map-reduce operation in the database side. The `map` and `reduce` functions are expressed as javascript methods.

Usage

```
mdbMapreduce(db, map, reduce, query = "{}", sort = "{}", limit = 0)

## S4 method for signature 'MongoDB'
mdbMapreduce(db, map, reduce, query = "{}", sort = "{}", limit = 0)

## S4 method for signature 'mongo'
mdbMapreduce(db, map, reduce, query = "{}", sort = "{}", limit = 0)
```

Arguments

`db`	(or MongoDB mong) – The collection to operate on.
`map`	character – A javascript function to apply to each document.
`reduce`	character – A javascript function to summarize the result
`query`	character – A JSON query to slect part of the collection. See `\link{buildJQuery}` and `\link{mdbFind}`.
`sort`	charcter – JSON object giving sorting order for result set (see `\link{mdbFind}`).
`limit`	integer – maximum number of records to process

Details

The Mongo database manual suggests that aggregation piplelines have better performance than map-reduce. Starting in Mongo 5.0 map-reduce is depricated.

Value

data frame with results

See Also

`\link[mongolite]{mongo}` `\link{mdbAggregate}` https://www.mongodb.com/docs/manual/core/map-reduce/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
histdata <- mdbMapreduce(irisdb,
  map= "function (){emit(Math.floor(this.Petal_Length*5)/5, 1)}",
  reduce="function (id,counts){return Array.sum(counts)}"
)
if (any(!is.na(histdata))) {
names(histdata) <- c("Petal.length","count")
}
head(histdata)
histdata1 <- mdbAggregate(irisdb,
 '[{"$set": {
    "ptlround": {
      "$divide":[
           {"$floor": {
            "$multiply": ["$Petal_Length", 5]
            }}, 5]}}},
   {"$group": {
     "_id": "$ptlround",
     "count": {"$sum":1}
   }}
]'
)
if (!is.null(histdata1)) {
names(histdata1) <- c("Petal.length","count")
}
head(histdata1)
```

---

mdbRemove *Remove selected objects from collection*

Description

Query selects a subset of the collection to remove. Note for removing everything, `\link{mdbDrop}` is faster.

Usage

```
mdbRemove(db, query = "{}", just_one = FALSE)

## S4 method for signature 'MongoDB'
mdbRemove(db, query = "{}", just_one = FALSE)

## S4 method for signature 'mongo'
mdbRemove(db, query = "{}", just_one = FALSE)
```

Arguments

db	(or MongoDB mongo) – Collection affected.
query	character – Mongo query expressed as JSON object. See \link{buildJQuery} and \link{mdbFind}.
just_one	logical – If true, only the first matching record is removed.

Value

miniprint Information about the results.

See Also

\link[mongolite]{mongo}, \link{mdbFind}, \link{mdbDrop} https://www.mongodb.com/docs/manual/reference/command/delete/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbDrop(irisdb)
mdbInsert(irisdb,iris)
mdbCount(irisdb)
mdbRemove(irisdb,'{"Species":"setosa"}')
mdbCount(irisdb)
```

---

mdbRename *Renames collection or moves it to a new database*

Description

Using the name argument simply renames the collection. Using the db argument copies the collection to a new database.

Usage

```
mdbRename(mdb, name, db = NULL)

## S4 method for signature 'MongoDB'
mdbRename(mdb, name, db = NULL)

## S4 method for signature 'mongo'
mdbRename(mdb, name, db = NULL)
```

Arguments

mdb	(or MongoDB mongo) – Reference to collection to move
name	character – new name for collection
db	character – new database for collection.

Value

miniprint Status Message

See Also

\link[mongolite]{mongo} https://www.mongodb.com/docs/manual/reference/command/renameCollection/

Examples

```
mdbDrop(MongoDB("FisherIrises",noMongo=!interactive()))
irisdb <- MongoDB("iris",noMongo=!interactive())
showCollections(irisdb)
mdbRename(irisdb,"FisherIrises")
showCollections(irisdb)
```

---

mdbReplace *Replace a document with a new document*

Description

Replace a document with a new document

Usage

```
mdbReplace(db, query, update = "{}", upsert = FALSE)

## S4 method for signature 'MongoDB'
mdbReplace(db, query, update = "{}", upsert = FALSE)

## S4 method for signature 'mongo'
mdbReplace(db, query, update = "{}", upsert = FALSE)

mdbUpsert(db, query, update = "{}", upsert = TRUE)

## S4 method for signature 'JSONDB'
mdbUpsert(db, query, update = "{}", upsert = TRUE)
```

Arguments

db	(or MongoDB mongo) Reference to the collection.
query	character Query as JSON document, see \link{mdbFind} and \link{buildJQuery}.
update	character Replacement document in JSON format.
upsert	logical If TRUE document will be inserted if the query does not return a result.

Details

In this method, the entire selected document (including the _id field is replaced. In the \link{update} method, the existing record is modified.

The query argument should return 0 or 1 arguments. If it returns 0 and upsert is TRUE, then the document is inserted. The function mbdUpsert(...) is an alias for mdbReplace(..., upsert=TRUE).

Value

miniprint with results.

See Also

\link[mongolite]{mongo}, \link{mdbFind}, \link{mdbUpdate} https://www.mongodb.com/docs/manual/reference/method/db.collection.replaceOne/

Examples

```
testdb <- MongoDB(noMongo=!interactive())
mdbDrop(testdb)
mdbInsert(testdb,'{"name":"Fred", "gender":"M"}')
mdbFind(testdb,fields='{}')
mdbReplace(testdb,'{"name":"Fred"}', '{"name":"Phred",
"gender":"F"}')
mdbFind(testdb,fields='{}')
```

---

mdbRun — *Runs a Mongo command on the collection*

Description

Runs a Mongo command on the collection

Usage

```
mdbRun(db, command = "{\"ping\":1}", simplify = TRUE)

## S4 method for signature 'MongoDB'
mdbRun(db, command = "{\"ping\":1}", simplify = TRUE)

## S4 method for signature 'mongo'
mdbRun(db, command = "{\"ping\":1}", simplify = TRUE)
```

Arguments

db	(or MongoDB mongo) Reference to the collection.
command	character JSON document providing the command.
simplify	logical If true, the output structure is simplified.

Details

A command is a JSON document. See the Mongo reference manual for the supported commands (these will vary quite a lot by the version of the database). Note that some commands only run against the "admin" database.

Value

list containing returned value.

See Also

`\link[mongolite]{mongo}` https://www.mongodb.com/docs/manual/reference/command/

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
mdbRun(irisdb,'{"collStats":"iris"}')
```

---

mdbUpdate *Modify document(s) in a collection*

Description

The `query` field identifies a number of documents. The `update` is a set of instructions for changing the documents. The `\link{mdbReplace}` function replaces the document instead of modifying it.

Usage

```
mdbUpdate(
  db,
  query,
  update = "{\"$set\":{}}",
  filters = NULL,
  upsert = FALSE,
  multiple = FALSE
)

## S4 method for signature 'MongoDB'
mdbUpdate(
  db,
  query,
  update = "{\"$set\":{}}",
  filters = NULL,
  upsert = FALSE,
  multiple = FALSE
)
```

```
## S4 method for signature 'mongo'
mdbUpdate(
  db,
  query,
  update = "{\"$set\":{}}",
  filters = NULL,
  upsert = FALSE,
  multiple = FALSE
)
```

Arguments

db	(or MongoDB mongo) – The collection to modify.
query	character – A JSON document identifying the document(s) to modify.
update	character – A JSON document identifying the changes to make to a document.
filters	character – A JSON document which controls which documents in the collection gets updated. (See the "Mongolite User Manual").
upsert	logical – If true and the query returns no results, then insert the document instead.
multiple	logical – If true, all documents matching query are affected; if false, only the first one.

Details

Queries:

The rules here match those for \link{mdbFind}.

Update Documents:

This is a special document which describes how to modify the document. There are a number of commands described in the Mongo documentation, of which the two most useful are $set and $unset.

The $set command takes as argument a JSON object giving field value pairs. If the field exists it value will be changed, if it doesn't exist a new field will be created. For example '{"$set":{"processed":false}}' will set the processed field to false, additing it if necessary.

The $unset command removes a field from a document. For example, '{"$unset":{"processed":0}}' removes the process field. The value after the field name is ignored.

For multiple complex changes, consider using 'mdbAggregate for more complex changes.

Value

A list with the returned object.

See Also

\link[mongolite]{mongo}, \link{mdbReplace} \link{mdbFind}, \link{mdbAggregate}

Examples

```
 mdb <- MongoDB("testthis","test","mongodb://localhost",noMongo=!interactive())
 mdbDrop(mdb)
 mdbInsert(mdb,c('{"name":"Fred", "gender":"M"}',
                 '{"name":"George", "gender":"M"}'))
mdbFind(mdb)
mdbUpdate(mdb,'{"name":"Fred"}', '{"$set":{"gender":"F"}}')
mdbFind(mdb)
```

MongoDB-class — *MongoDB – Reference class wrapping a connection to a Mongo database collection.*

Description

MongoDB – Reference class wrapping a connection to a Mongo database collection.

Usage

```
MongoDB(
  collection = "test",
  db = "test",
  url = "mongodb://localhost",
  verbose = FALSE,
  noMongo = FALSE,
  options = mongolite::ssl_options()
)
```

Arguments

collection	character – name of collection
db	character – name of database
url	character – URI for mongo connection (see makeDBuri())
verbose	logical – Should operate in verbose mode.
noMongo	logical – If true, then no connection to Mongo database will be made, and CRUD operations will become no-ops.
options	– SSL options for connections, see mongolite::ssl_options().

Details

Including a mongo object in an Reference class presents a potential race condition. The prototype class is built at package load time, however, calling the \link[mongolite]{mongo} may not work at this time. The MongoDB class works around this by capturing the arguments to the mongo call, and then creating the actual database connection when the database is first accessed. The database should always be accessed through the $db() method which builds the database if needed.

Value

An object of class MongoDB

Fields

mongoObj ANY – This is the actual \link[mongolite]{mongo} object or NULL if it has not been initialized yet.

uri character – URI for the mongo connection.

dbname character – The name of the mongo database.

colname character – The name of the mongo collection

noMongo logical – If TRUE, then the This allows a class which contains a reference to a Mongo database to ignore the database calls when there is no database to connect to.

verbose logical – This field is passed on to the \link[mongolite]{mongo} call.

options ANY – This field is passed on to the \link[mongolite]{mongo} call. It is used to store additional SSL connection information, see \link[mongolite]{ssl_options}.

Class-based Methods

- $initialize(collection, db, url, verbose, options, ...) – Constructor.
- $db() – Returns the the actual database connection (\link[mongolite]{mongo} object), or NULL if uri=="" or noMongo==TRUE. If the actual call to 'mongo has not been made, this method will create the connection; otherwise, the cached connection is returned.
- $available() – Returns false if no database is present (i.e., noMongo is TRUE. Used to suppress actual mongo calls when database is not available.
- $resetDB() – Resets the mongoObj field to force a reconnection to Mongo the next time $db() is called. This is probably useful to call when restoring an R session.
- $toString() – Returns a string represenation of an object.

Methods

The S4 generic functions correspond to the normal CRUD (Create, Read, Update and Delete) methods. Particularly: \link{mdbAggregate}, \link{mdbCount}, \link{mdbDisconnect}, \link{mdbDrop}, \link{mdbExport}, \link{mdbFind}, \link{mdbImport}, \link{mdbIndex}, \link{mdbInsert}, \link{mdbIterate}, \link{mdbMapreduce}, \link{mdbRemove}, \link{mdbRename}, \link{mdbReplace}, \link{mdbRun}, \link{mdbUpdate}, \link{showCollections} and \link{showDatabases}.

Note

Many of the examples use MongoDB(...,noMongo=!interactive()). This means the dummy mechanism will be used during package checking (where Mongo may or may not be available in the development environment), but running the examples from the help files will make the connections (and will generate an error if Mongo is not installed).

See Also

[mongo] More extensive documentation on most of the mdbXXX functions can be found at the Mongo API documentation web site. https://www.mongodb.com/docs/manual/reference/command/

Examples

```
mdp <- MongoDB("test","test","mongodb://localhost")
## Not run:
# This will generate an error if mongo doesn't exist.
mdbCount(mdp,'{}')

## End(Not run)
nullmdp <- MongoDB(noMongo=TRUE)
mdbCount(nullmdp)
# This will return `NA`.
```

MongoRec-class *Class "MongoRec".*

Description

This is a lightweight class meant to be extended. It contains a single field for a Mongo identifier, which can be accessed using the `m_id()` method. It is meant to store something that is a record in a Mongo collection, where `_id` is the Mongo identifier.

Usage

```
MongoRec(..., m_id = NA_character_)
```

Arguments

`...`	Other arguments (pass through for initialization method).
`m_id`	character Mongo identifier. Use `NA_character_` for unsaved objects.

Value

The constructor `MongoRec` returns an object of class `MongoRec`.

The `m_id` method returns a character scalar (with name `oid`) which contains the mongo identifer.

Functions

- `MongoRec()`: Constructor for `MongoRec`

Slots

`_id` (character) The Mongo ID, `NA_character_` if not saved.

Objects from the Class

Objects can be created by calls to the `MongoRec()` function.

Author(s)

Russell G. Almond

See Also

as.json() buildObject(), saveRec(), getOneRec()

Examples

```
showClass("MongoRec")
```

---

m_id *Accessor for the Mongo id element of a record.*

Description

Objects of class MongoRec have a _id slot which stores the database ID. This function accesses it.

Usage

```
m_id(x)

m_id(x) <- value

## S4 method for signature 'MongoRec'
m_id(x)

## S4 replacement method for signature 'MongoRec'
m_id(x) <- value
```

Arguments

x	An object of type MongoRec.
value	(character) the new ID value, use NA_character_ for missing.

Details

The _id slot should be a character object with the name "oid". The methods enforce this. If the object does not have a Mongo ID (i.e., it was never stored in a database), then the value of _id should be NA_character_.

Functions

- m_id(x) <- value: Setter for Mongo ID

Examples

```
mr <- MongoRec()
m_id(mr) # NA
m_id(mr) <- "012345"
m_id(mr)
```

`parse.jlist` *Construct an S4 object from a list of its slot values.*

Description

The `parse.json` function uses the `fromJSON` function to turn the JSON into a list, which is processed using the function `parse.jlist` to massage the elements, and then passes it to the `new` function to create a new object of type `class`.

Usage

```
parse.jlist(class, rec)

## S4 method for signature 'ANY,list'
parse.jlist(class, rec)

buildObject(rec, class = decodeClass(rec$class))

parse.json(encoded, builder = buildObject)

## S4 method for signature 'MongoRec,list'
parse.jlist(class, rec)
```

Arguments

`class`	– A character string defining the class of the output object. If the list has an element named `class`, that will be used.
`rec`	– A list which is the output of `fromJSON`
`encoded`	– a character scalar giving the raw JSON object.
`builder`	– A function which will construct an object from a list of fields values.

Details

The `parse.jlist` function is a helper function designed to do any massaging necessary to unencode the slot values before the object is produced. The function `ununboxer` undoes the effect of `unboxer`, and the function `unparseData` undoes the effect of `parseData`.

Value

An S4 object of type `class`

Functions

- `parse.jlist()`: This is the inner function for processing the slots prior to object creation. Generally, this is the method that needs to be specialized. See the `vignette("JSON for S4 Objects")`.
- `parse.jlist(class = ANY, rec = list)`: Base case for callNextMethod; just returns the slot list.
- `buildObject()`: This method takes the jlist, cleans it with an appropriate `parse.jlist` method and then tries to generate an object based on the class.
- `parse.jlist(class = MongoRec, rec = list)`: Makes sure the `_id` field corresponds to conventions, and inserts `NA` if it is missing.

Examples

```
## Not run:
vignette("JSON for S4 Objects")

## End(Not run)
```

---

parseData — *Prepare R data for storage or restore R data from jlist #' The* `parseData` *function is a helper function for* `parse.jlist()` *methods, and* `unparseData` *for* `as.jlist()`*, which represents complex objects as JSON.*

Description

Prepare R data for storage or restore R data from jlist #' The `parseData` function is a helper function for `parse.jlist()` methods, and `unparseData` for `as.jlist()`, which represents complex objects as JSON.

Usage

```
parseData(messData)

unparseData(data, serialize = TRUE)
```

Arguments

`messData`	(or character jlist)
`data`	ANY the data to be saved.
`serialize`	logical if Tru

Details

There are three strategies for saving/restoring an R object a JSON.

- Use the \link[jsonlite]{serializeJSON} and \link[jsonlite]{unserializeJSON} method. This will faithfully reproduce the object, but it will be difficult to manipulate the object outside of R.
- For an S4 object write a `as.jlist()` and `parse.jlist()` method.
- For a S3 object or just a list of arbitrary objects, write out the object using \link[jsonlite]{toJSON} and fix up the types of the components when the object is read back in.

When `unparseData(...,serialize=TRUE)` is called, then `parseData` and `unparseData` take the first approache.

Otherwise, it takes the third approach. In particular, \link[jsonlite]{fromJSON} turns a vector which contains all elements of the same type (currently only "logical", "integer", "numeric" and "character") it turns the list into a vector of the corresponding mode.

Value

`parseData` returns the parsed object. `unparseData` returns a jlist or character scalar which can be saved.

Examples

```
dat <- list(chars=letters[1:3], nums=c(-3.3, 4.7),
  ints=1L:3L, logic=c(TRUE,FALSE))
j1 <- jsonlite::toJSON(unparseData(dat))
j2 <- unparseData(dat,serialize=TRUE)
jsonlite::fromJSON(j1)
parseData(jsonlite::fromJSON(j1))
parseData(jsonlite::fromJSON(j2))
```

`parsePOSIX` *Convert Mongo dates to POSIX*

Description

Converting a date to Mongo-flavored JSON produces a numeric value (number of second seconds since Jan 1, 1970) labeled with `$date`. This function takes the output of \link[jsonlite]{fromJSON} and converts it back to POSIX format.

Usage

```
parsePOSIX(x)
```

Arguments

x	– Either a list of the form `list("$date"=1234)` or any other value compatible with `link[base]{as.POSIXct}`.

Details

If the date has been marked as scalar (using the `\link{unboxer}` or `\link[jsonlite]{unbox}`, this function will strip the `scalar` flag.

Value

the contents of `x` as a `POSIXct` object.

Examples

```
dt <- Sys.time()
dtj <- jsonlite::toJSON(unboxer(dt))
parsePOSIX(jsonlite::fromJSON(dtj,FALSE))
```

---

parseSimpleData *parseSimpleData*

Description

Simple parser works with data which is mostly primitive R types (`numeric`, `integer`, `logical`, `character`).

Usage

```
parseSimpleData(messData)
```

Arguments

`messData`	list output from `\link[jsonlite]{fromJSON}`

Details

The `\link[jsonlite]{fromJSON}` method does not distinguish between arrays type character, logical, integer or numeric. This function finds lists of a single atomic type and replaces them with the corresponding `vector`.

Value

list, simplified.

Examples

```
parseSimpleData(list(chars=list("a","b","c"),nums=list(2.3,3.4,4.5),
  ints=list(1,2,3), logic=list(TRUE,FALSE)))
```

saveRec *Saves a MongoRec object to a Mongo database*

Description

This function saves an S4 object as a record in a Mongo database. It uses `as.json` to covert the object to a JSON string.

Usage

```
saveRec(col, rec, serialize = TRUE)
```

Arguments

`col`	(or MongoDB mongo NULL) A mongo collection reference. If `NULL` record will not be saved.
`rec`	The message (object) to be saved.
`serialize`	A logical flag. If true, `serializeJSON` is used to protect the `data` field (and other objects which might contain complex R code.

Value

Returns the message argument, which may be modified by setting the `"_id"` field if this is the first time saving the object.

Author(s)

Russell Almond

See Also

`as.json`, `MongoRec`, `buildObject`, `getOneRec`, `MongoDB`

Examples

```
## Not run:
load_Events() # Uses the sample Event class.
m1 <- new("Event",uid="Fred",mess="Task Done",
                timestamp=Sys.time(),
                data=list("Selection"="B"))
m2 <- new("Event",uid="Fred",mess="New Obs",timestamp=Sys.time(),
                data=list("isCorrect"=TRUE,"Selection"="B"))
m3 <- new("Event",uid="Fred","New Stats",
                details=list("score"=1,"theta"=0.12345,"noitems"=1))

testcol <- MongoDB("Messages",noMongo=!interactive())
## Save them back to capture the ID.
m1 <- saveRec(testcol,m1)
```

```
m2 <- saveRec(testcol,m2)
m3 <- saveRec(testcol,m3)

## End(Not run)
```

---

showCollections *Shows collections in the current database.*

Description

Shows collections in the current database.

Usage

```
showCollections(
  db = NULL,
  dbname = "test",
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)

## S4 method for signature 'mongo'
showCollections(
  db = NULL,
  dbname = "test",
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)

## S4 method for signature 'MongoDB'
showCollections(
  db = NULL,
  dbname = "test",
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)

## S4 method for signature 'NULL'
showCollections(
  db = NULL,
  dbname = "test",
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)
```

Arguments

db	(or MongoDB mongo NULL) – Connection to target database. If NULL a new connection is made using the specified information.
dbname	character – name for new collection
uri	character – URI for database connections.
options	list – SSL options for an SSL connection

Details

Shows all of the collections which are in the referenced database. If the db argument is a MongoDB or mongolite::mongo object, the current database is used. If db is NULL, then a new connection is created with the information.

Value

character vector of database names

See Also

[mongo]

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
showCollections(irisdb)
```

---

showDatabases *Lists Databases*

Description

Lists Databases

Usage

```
showDatabases(
  db = NULL,
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)

## S4 method for signature 'MongoDB'
showDatabases(
  db = NULL,
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)
```

```
## S4 method for signature 'NULL'
showDatabases(
  db = NULL,
  uri = "mongodb://localhost",
  options = mongolite::ssl_options()
)
```

Arguments

db	(or MongoDB NULL) – Database reference. (Note: `mongolite::mongo` cannot be used.)
uri	character – URI for database connections.
options	list – SSL options for an SSL connection

Details

This function lists the names of the databases which are accessible for the current user.

This function needs to make a new connection to the `admin` database. If a MongoDB object is supplied, then the `uri` and `options` are taken from it. If the `db` argument is NULL, a new connection is made.

Note that there is currently no documented way of retrieving the `url` and `ssl_options` from a `mongolite::mongo` object.

Value

a data frame containing the names and sizes of the databases

See Also

`\link[mongolite]{mongo}`, `\link{mdbRun}`

Examples

```
irisdb <- MongoDB("iris",noMongo=!interactive())
showDatabases(irisdb)
```

---

unboxer *Marks scalar objects to be preserved when converting to JSON*

Description

The function `toJSON` coverts vectors (which all R objects are) to vectors in the JSON code. The function `jsonlite::unbox` protects the object from this behavior, which makes the fields eaiser to search and protects against loss of name attributes. The function `unboxer` extents `unbox` to recursively unbox lists (which preserves names). The function `ununbox` removes the unboxing flag and is mainly used for testing parser code.

Usage

```
unboxer(x)

ununboxer(x)
```

Arguments

x	Object to be boxed/unboxed.

Details

The `jsonlite::unbox` function does not necessarily preserve the name attributes of elements of the list. In other words the sequence `as.jlist` -> `toJSON` -> `fromJSON` -> `buildObject` might not be the identity.

The solution is to recursively apply `unbox` to the elements of the list. The function `unboxer` can be thought of as a recursive version of `unbox` which handles the entire tree struction. If `x` is not a list, then `unboxer` and `unbox` are equivalent.

The typical use of this function is defining methods for the `as.jlist` function. This gives the implementer fine control of which attributes of a class should be scalars and vectors.

The function `ununboxer` clears the unboxing flag. Its main purpose is to be able to test various parsers.

Value

The function `unboxer` returns the object with the added class `scalar`, which is the `jsonlite` marker for a scalar.

The function `ununboxer` returns the object without the `scalar` class marker.

Functions

- `ununboxer()`: Undoes the effect of unboxer (in particular, removes the scalar mark).

Warning

Dependence on jsonlite implementation:

These functions currently rely on some internal mechanisms of the jsonline pacakge. In particular, `ununbox` relies on the "scalar" class mechanism.

Note

There is a bug in the way that `POSIXt` classes are handled, `unboxer` fixes that problem.

Author(s)

Russell Almond

See Also

unbox, toJSON, as.jlist, buildObject

Examples

```
## Not run:
load_examples() ## Example uses event class.

## as.jlist method shows typical use of unboxer.
getMethod("as.jlist",c("Event","list"))

## Use ununboxer to test as.jlist/buildObject pair.
m4 <- Event("Phred","New Stats",
               data=list("agents"=c("ramp","ramp","lever")))
m4jl <- as.jlist(m4,attributes(m4))
m4a <- buildObject(ununboxer(m4jl))
testthat::expect_equal(m4a,m4)

## End(Not run)
```

Index