Package: Proc4 (via r-universe)

September 9, 2024

Version 0.8-5

Date 2023/06/23

Title Four Process Assessment Database and Dispatcher

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), methods,

Imports futile.logger, mongo, jsonlite

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

Description Utilities for working with messages in the four four process architecture as json objects.

Collate ErrorHandling.R Message.R MessageQueue.R Listeners.R CaptureListener.R InjectionListener.R TableListener.R UpdateListener.R UpsertListener.R

License Artistic-2.0

URL https://pluto.coe.fsu.edu/Proc4

VignetteBuilder knitr

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 Science 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')

Config/testthat/edition 3

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

RemoteUrl https://github.com/ralmond/Proc4

RemoteRef HEAD

RemoteSha b45286c302d5e2e795c41387d1c85ccabcbec0a2

Contents

`Proc4-package` *Four Process Assessment Database and Dispatcher*

Description

Utilities for working with messages in the four four process architecture as json objects.

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 exists to supply core functionality to other processes implementing processes in the four process architecture (Almond, Steinberg and Mislevy, 2002). In particular, it contains low level code dealing with implementing message queues in a document database (mongo) and reading/writing messages from JSON.

There are five major features of this package documented below:

1. The `P4Message` object and the protocol for converting messages to JSON and saving them in the mongo database.
2. A `withFlogging` function which wraps the `flog.logger` protocol.
3. A number of `Listener` objects which implement an observer protocol for messages.
4. The `config` directory contains a number of javascript files for building database schemas and indexes.
5. The `dongle` directory contains a number of PHP scripts for exposing the database via a web server.

Earlier verisons included a number of tools which wrap funcions in the `mongolite` and `jsonlite` packages. In particular, this included the `as.json`, `parse.json`, and `buildObject` generic functions to manage the conversion from S4 object to JSON and back, the `saveRec`, `getOneRec`, and `getManyRecs` methods for saving and restoring objects from a database, and the `buildJQuery` for building Mongo queries from R-like syntax. These have been moved to the `mongo` package.

P4 Messages

The extended four process architecture defines a message object (`P4Message`) with the following fields:

`_id`: Used for internal database ID.

`app`: Object of class `"character"` which specifies the application in which the messages exit.

`uid`: Object of class `"character"` which identifies the user (student).

`context`: Object of class `"character"` which identifies the context, task, or item.

`sender`: Object of class `"character"` which identifies the sender. This is usually one of "Presentation Process", "Evidence Identification Process", "Evidence Accumulation Process", or "Activity Selection Process".

`mess`: Object of class `"character"` a general title for the message context.

`timestamp`: Object of class `"POSIXt"` which gives the time at which the message was generated.

`data`: Object of class `"list"` which contains the data to be transmitted with the message.

`processed`: A logical value: true if the message has been processed, and false if the message is still in queue to be processed. This field is set with `markAsProcessed`.

pError: If a error occurs while processing this event, information about the error can be stored here, either as an R object, or as an R object of class error (or any class). This field is accessed with processingError and set with markAsError.

Other classes can extend this message protocol by adding additional fields, but the header fields of the message object allow it to be routed.

In particular, the processed field allows a database collection of messages to be used as queue. Simply search for unprocessed message and begin processing them oldest first, using markAsProcessed to mark the complete process and markAsError to mark errors.

The mongo::as.json and mongo::parse.json functions build JSON representations of S classes. In general, this process needs explicit instructions on how to code/decode the fields of the object. Methods of the inner mongo::as.jlist and mongo::parse.jlist provide this functionality. Note that classes which extend P4Message class will need to use these methods. The cleanMessageJlist does the common processing for the P4Message parent class. Finally, buildMessage is a more specific version of the buildObject generic builder.

The functions saveRec, getOneRec and getManyRecs facilitate saving and loading message objects from the database. The function buildJQuery gives R-like syntactic sugar to building mongo (JSON) queries.

Logging

The logging system for the Proc4 processes is mostly just the flog.logger protocol. Aside from importing the futile.logger package, Proc4 adds the function withFlogging executes a series of statements in an environment in which the error messages will be logged, and at higher logging levels, stack traces for errors and warnings are given. The intention is that most message handling functions will be wrapped in withFlogging, so that information about the message causing the error/warning will be available for debugging.

The package also supplies a mongoAppender class, which provides a way of logging messages to a database.

Listeners

The Proc4 package implements an observer protocol called Listener. A listener is an abstract class which implements the receiveMessage function. The argument of this function is a P4Message object, which the listener then does something with. (In most of the implemented examples, this is to save it in a database.) Note that listeners should also define a isListener method to indicate that it is a listener.

Four listeners are currently implemented (see Listener or the individal listener classes):

CaptureListener Creates an object of class CaptureListener which stores the messages in a list.

InjectionListener Creates an object of class InjectionListener which inserts the message into the designated database.

UpdateListener Creates an object of class UpdateListener which updates the designated field.

UpsertListener Creates an object of class UpsertListener which insert or replaces the message in the designated collection.

`TableListener` Creates an object of class `TableListener` which adds details from message to rows of a data frame.

The `RefListener` is an abstract class which provides methods for the other classes (in particular, promoting the class-based methods to true S4 methods. These include `isListener()`, `listenerName`, `listeningFor`, `receiveMessage`, and `clearMessage`. Note that the default methods for the latter two functions rely on internal `$receiveMessage()` and `$reset()` class-based methods, which must be implemented in the subclasses.

The `ListenerSet` class is a mixin to associate a collection of listeners with an object (the `EIEngine` and `BNEngine` classes use this). The generic function `notifyListeners` can be called. This logs information about the message (see logging system above), save a copy of the message in a "Messages" database, and calls the `receiveMessage` method on all of the listener objects in its collection.

Configuration Files

Using the mongo database, both security (user IDs and passwords) is optional. Running mongo without security turned on is probably okay as long as the installation is (a) behing a firewall, and (b) the firewall is configured to not allow connections on the mongo port except from localhost. However, other users may want to turn on security.

The recommended security setup is to create four users, "EIP", "EAP", "ASP", and "C4" for the four processes and to assign a password to each. The URI's of the database connections then need to be modified to include the username and passwords. Each process would have an `ini.R` file which contains its password which is stored in an appropriate configuration directory. (On *nix systems, the recommend location is `/usr/local/share/Proc4`.)

The files `Proc4.ini` (PHP format) and `Proc4.js` (javascript format) can be used for saving the key usernames and passwords. These files are located in the directory `file.path( library(help="Proc4")$path, "config")`. To install these files it is necessary to copy the files to the configuration directory and edit them so that the password reflects local preferences.

The file `setupDatabases.js` in the `config` directory creates databases for each of the processes and stores the appropriate login credentials. (Note that this calls `Proc4.js` to get these credentials so that file must be established first.) This is a javascript file designed to be run directly in mongo, i.e., `mongo setupDatabases.js`. Note that it must be run by a user which has the appropriate priveleges to create databases and modify their security (a "root" user).

The file `setupProc4.js` in the `config` directory sets up schemas and indexes for collections in the `Proc4` database which are used by the dongle process. Schemas are optional in mongo, but the indexes should speed up operations.

Dongle Files

The directory `file.path(library(help="Proc4")$path, "config")` contains files that facilitate direct communciation with the mongo database. In particular, there are a number of PHP scripts which if put in a directory available to the web server will allow remote processes to get information about users in the system. The scripts are:

`PlayerStart.php` Called when player logs in on a given day. As data returns information needed to restore gaming session (currently bank balance and list of trophies earned). Note that player details are updated by the EI process.

`PlayerStop.php` Called when player logs out. Currently not used. It is designed to help automatically shut down unneeded processed.

`PlayerStats.php` Called when current player competency estimates are required, e.g., when displaying player scores. It returns a list of statistics and their values in the data field; the exact statistics returned depend on the configuration of the EA process. This database collection is updated by the EA process after each game level is processed.

`PlayerLevels.php` Called when the game wants the next level. The message data should contain information about what topic the player is currently addressing and a list of played and unplayed levels, with the unplayed levels sorted so the next level according to protocol is first on the list. The complete list of levels should be returned so that if levels on the list have already been completed, a new level would be entered. Although the PHP script has been built, the AS process to feed it has not.

In addition, there is a file called `LLtoP4` in that directory which is a `bash` script for translating between xAPI and Proc4 message formats. The function `LLtoP4Loop` repeatedly downloads xAPI statements from the learning locker database, translates them to P4 format, and uploads them to the EI process database.

The vingette file `Dongle.pdf` describes the dongle and database structure in more detail.

Acknowledgements

Work on the Proc4, EIEvent and EABN packages has been supported by the following grants:

- Bill and Melinda Gates Foundation grant "Games as Learning/Assessment: Stealth Assessment" (no. 0PP1035331, Val Shute, PI)
- National Science Foundation grant "DIP: Game-based Assessment and Support of STEM-related Competencies" (no. 1628937, Val Shute, PI)
- National Scient Foundation grant "Mathematical Learning via Architectual Design and Modeling Using E-Rebuild." (no. 1720533, Fengfeng Ke, PI)
- Institute of Educational Statistics Grant: "Exploring adaptive cognitive and affective learning support for next-generation STEM learning games." (no. R305A170376-20, Val Shute and Russell Almond, PIs')

The Proc4 package developement was led by Russell Almond (Co-PI).

Author(s)

Russell Almond

Maintainer: Russell Almond <ralmond@fsu.edu>

References

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

The source code, and issues database can be found at https://github.com/ralmond/Proc4

See Also

flog.logger, EIEvent, EABN

---

buildListener *Builds a listener from a JSON description.*

Description

This is used in configuration, it will build a listener from a JSON description of the listener. The "name" and "type" fields are required. The other fields should match the arguments for the constructor, with the exceptions noted below:

Usage

```
buildListener(specs, app, dburi, defaultDB="Proc4",
              ssl_options=mongolite::ssl_options(),
           noMongo = !missing(dburi) && length(dburi) > 0L && nchar(dburi) > 0L)
```

Arguments

specs	A named list (from the JSON) containing the instructions for building the listener.
app	A character value that will get substituted for the string "<app>" in the "name" and "sender" fields.
dburi	If a database is used for this listener, then this is the uri for the connection. Note that this is specified in the code and not in the JSON.
defaultDB	The name of the database with which the Listener will interact, only used if no dbname field in specs.
ssl_options	Options used for an SSL connection to the database. ssl_options.
noMongo	A logical value. If true, then the connection to the Mongo database will not be made, and CRUD operations will basically become no-ops.

Details

The input to this function is a list that comes from JSON (or some other input method that returns a named list). The specs$type field should be the name of a Listener class. This means that specs$type is the name of a constructor function, and the rest of the spec argument are the arguments.

Currently, the following fields are used.

name The name of the listener, required. The string "<app>" is substituted for app.

type Required, the name of the constructor for the desired class. The function will generate an error if this does not correspond to the name of a class.

sender A string insterted into logged messages. The string "<app>" is substituted for app.

dbname The name of the database in which the messages will be recorded. If not present, then the `defaultdb` will be used.

colname The name of the database collection in which the messages will be recorded.

messages A character vector giving the names of the messages the listener will pay attention to. Note that this maps to the field "messSet" in the listener object.

targetField Used in the `UpdateListener` and `UpsertListener` to indicate the field to be modified.

jsonEncoder The name of a function used to encode the field value to be modified as JSON. See `stats2json`.

qfields A character vector giving the names of the fields used as the key for finding the message to replace. Usually should contain `c("uid","app")`.

fields This should be a named character vector (or list) whose names indicate the names of the observables/statistics to collect, and whose values are the types. See `TableListener`; this field maps to the "fieldlist" field of that class.

Other fields in `specs` are ignored.

Value

An object of the virtual class `Listener` (i.e., something for which `isListener` should return true.

Note

The field name "messages" maps to the internal field `messSet`. The field name "fields" maps to the internal field `fieldlist`.

Author(s)

Russell Almond

See Also

`Listener`, `fromJSON`

Examples

```
jspecs <- '[
 {
   "name":"ppLS<app>",
   "type":"TableListener",
   "messages":["Coins Earned","Coins Spent", "LS Watched"],
   "fields":{
    "uid":"character",
    "context":"character",
    "timestamp":"character",
    "currentMoney":"numeric",
    "appId":"numeric",
    "mess":"character",
    "money":"numeric",
    "onWhat":"character",
```

```
    "LS_duration":"difftime",
    "learningSupportType":"character"
   }
 },
 {
     "name":"ToEA",
     "type":"InjectionListener",
     "dbname":"EARecords",
     "colname":"EvidenceSets",
     "messages":["New Observables"]
 },
 {
     "name":"PPPersistantData",
     "type":"UpdateListener",
     "dbname":"Proc4",
     "colname":"Players",
     "targetField":"data",
     "jsonEncoder":"trophy2json",
     "messages":["Money Earned", "Money Spent"]
 }
    ]'

speclist <- jsonlite::fromJSON(jspecs,FALSE)

l1 <- buildListener(speclist[[1]],"test",mongo::makeDBuri())

l2 <- buildListener(speclist[[2]],"test",mongo::makeDBuri())

l3 <- buildListener(speclist[[3]],"test",mongo::makeDBuri())
```

`buildListenerSet` *Builds Listener Set from a a JSON configuration*

Description

This method builds a `ListenerSet` for an engine. In particular, the `config` is list which come from reading a JSON file (see `fromJSON`) which contains the rules for building the `Listener`s in the set.

Usage

```
buildListenerSet(sender, config, appid, lscol, dbname, dburi,
                 sslops, registrycol, registrydbname,
                 mongoverbose = FALSE)
```

Arguments

`sender`	A character scalar identifying the message sender.

config	A named list providing details of the contained listeners.
appid	A character scalar giving the application ID for the application being built.
lscol	A character scalar giving the name of the collection used for logging messages by the message set.
dbname	A character scalar giving the name of the database for the message log, as well as the default database for listeners.
dburi	A character scalar giving the URI of the mongo collection.
sslops	A list giving options for a SSL connection. See ssl_options.
registrycol	A character scalar giving the name of the colleciton for registering output.
registrydbname	A character scalar giving the name of the database in which the output registriation collection
mongoverbose	A flag for adding debugging information to Mongo calls (see MongoDB).

Details

This method builds the listener set starting by calling buildListener(config[[i]]) for each element of the config list. This then becomes the listeners to the ListenerSet constructor.

Note that the appid, dburi, dbnmae (mapped to defaultDB), and sslops are passed to buildListeners to use for defaults.

Value

An object of class ListenerSet.

Author(s)

Russell Almond

See Also

ListenerSet, buildListener

Examples

```
## Not run:
jspecs <- '{
     "listeners":[
 {
     "name":"ToAS",
     "type":"InjectionListener",
     "dbname":"ASRecords",
     "colname":"Statistics",
     "messages":["Statistics"]
 },
 {
     "name":"PPStats",
     "type":"UpdateListener",
     "dbname":"Proc4",
```

```
     "colname":"Statistics",
     "targetField":"data",
     "jsonEncoder":"stats2json",
     "messages":["Statistics"]
 }

   ]}'

speclist <- jsonlite::fromJSON(jspecs,FALSE)

lset <- buildListenerSet("TestEngine",speclist$listeners,
                         "ecd://pluto.coe.fsu.edu/P4Test",
                         lscol="Messages",dbname="test",
                         dburi="", sslops=mongolite::ssl_options(),
                         registrycol="OutputFiles",
                         registrydbname="test")

## End(Not run)
```

---

buildMessage *Converts a JSON object into a P4 Message*

Description

The buildMessage function is a parser to use with the getOneRec and getManyRecs database query functions. This function will convert the documents fetched from the database into P4Message objects.

Usage

```
buildMessage(rec,class="P4Message")
cleanMessageJlist(rec)
## S4 method for signature 'P4Message,list'
as.jlist(obj, ml, serialize = TRUE)
## S4 method for signature 'P4Message,list'
parse.jlist(class, rec)
```

Arguments

rec	A named list containing JSON data.
class	The class of the object being built. In the case of buildMessage, this can be the name of the class.
obj	The object being converted. This is mostly used for message dispatch.
ml	A named list containing JSON data.
serialize	If true, then the serializeJSON method is used to preserve the details field of the message.

Details

The mdbIterate method object returns a list containing the fields of the JSON object with a *name*=*value* format (see jlist). This is the rec argument to buildMessage. In particular, this is a builder function (see buildObject) which can be passed as the builder argument to getOneRec() or getOneRec() when the object to be built is a P4Message object.

To facilitate the building subclasses, the (e.g., to check the argument types and insert default values). The function cleanMessageJlist does that cleaning for the common fields of the P4Message object, so subclasses P4Message can inheret the parsing for the commond message fields. The as.jlist method is a helper function for the as.json method. The parse.jlist method (which calls cleanMessageJList is a helper function for the parse.json method.

The data field needs extra care as it could contain arbitrary R objects. There are two strategies for handling the data field. First, use serializeJSON to turn the data field into a slob (string large object), and unserializeJSON to decode it. This strategy should cover most special cases, but does not result in easily edited JSON output. Second, recursively apply unboxer and use the function parseSimpleData to undo the coding. This results in output which should be more human readable, but does not handle objects (either S3 or S4). It also may fail on more complex list structures.

Value

The function buildMessage returns a P4Message object populated with fields from the rec argument. The function cleanMessageJlist and the parse.jlist method returns the cleaned rec argument (suitable for passing to the P4Message constructor).

The function as.jlist method returns the processed ml object (ready to be converted to JSON).

Note

I hit the barrier pretty quickly with trying to unparse the data manually. In particular, it was impossible to tell the difference between a list of integers and a vector of integers (or any other storage type). So, I went with the serialize solution.

The downside of the serial solution is that it stores the data field as a slob. This means that data values cannot be indexed. If this becomes a problem, a more complex implementation may be needed.

Author(s)

Russell Almond

See Also

as.jlist, getOneRec, getManyRecs, P4Message

serializeJSON, unserializeJSON

Examples

```
m1 <- P4Message("Fred","Task1","PP","Task Done",
                details=list("Selection"="B"))
m2 <- P4Message("Fred","Task1","EI","New Obs",
                details=list("isCorrect"=TRUE,"Selection"="B"))
```

```
m3 <- P4Message("Fred","Task1","EA","New Stats",
                details=list("score"=1,"theta"=0.12345,"noitems"=1))

ev1 <- P4Message("Phred","Level 1","PP","Task Done",
      timestamp=as.POSIXct("2018-12-21 00:01:01"),
      details=list("list"=list("one"=1,"two"=1:2),"vector"=(1:3)))


m1a <- buildMessage(mongo::ununboxer(as.jlist(m1,attributes(m1))))
m2a <- buildMessage(mongo::ununboxer(as.jlist(m2,attributes(m2))))
m3a <- buildMessage(mongo::ununboxer(as.jlist(m3,attributes(m3))))

ev1a <- buildMessage(mongo::ununboxer(as.jlist(ev1,attributes(ev1))))
```

`CaptureListener-class` *Class* `"CaptureListener"`

Description

This listener simply takes its messages and adds them to a list. It is is mainly used for testing the message system.

Details

This listener simply takes all messages and pushes them onto the `messages` field. The `messages` field is the complete list of received messages, most recent to most ancient. The method `lastMessage()` returns the most recent message.

Extends

This class implements the `Listener` interface.

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

isListener `signature(x = "CaptureListener")`: returns true.

receiveMessage `signature(x = "CaptureListener")`: If the message is in the `messSet`, it adds the message to the message list. (See details)

listenerName `signature(x= "InjectionListener")`: Returns the name assigned to the listener.

listenerDataTable `signature(listener = "CaptureListener", appid )`: Builds a data datable from the messages.

Data Table

When the `listenerDataTable` method is called, the table is made by applying the `attributes` function to the `$messages` list. As these are presumably `P4Message` objects, this will expose the fields as a database.

Fields

`messages`: Object of class `list` the list of messages in reverse chronological order.

Class-Based Methods

`lastMessage()`: Returns the most recent message.

`receiveMessage(mess)`: Does the work of inserting the message. See Details.

`reset(app)`: Empties the message list.

`initialize(messages, ...)`: Sets the default values for the fields.

Author(s)

Russell Almond

References

This is an example of the observer design pattern. https://en.wikipedia.org/wiki/Observer_pattern.

See Also

Listener, P4Message, CaptureListener, UpdateListener, UpsertListener, InjectionListener, TableListener,

Examples

```
mess1 <- P4Message(app="default",uid="Phred",context="Down Hill",
                   sender="EABN",mess="Statistics",
                   details=list("Physics_EAP"=0.5237,"Physics_Mode"="High"))

cl <- CaptureListener()
receiveMessage(cl,mess1)
stopifnot(all.equal(mess1,cl$lastMessage()))
```

---

cleanMessageQueue *Removes messages matching query from queue.*

Description

Often a queue will contain a number of messages which do not get processed. This function cleans out messages from the queue. This is typically called both before (called "cleaning") importing new messages (see importMessages) and after (called "purging").

Usage

```
cleanMessageQueue(queue, query, appid)
## S4 method for signature 'MongoQueue'
cleanMessageQueue(queue, query, appid)
```

Arguments

queue	An object of class MessageQueue to be cleaned.
query	A list which forms a Mongo query for selecting the messages to be removed. See buildJQuery.
appid	A character scalar giving the name of the application to be cleaned.

Value

Return value is undefined. Called for its side effects (removing messages from database collection).

Will log and throw database errors.

Note

Generates log entries using ‘futile.logger‘.

Currently no method for ‘ListQueue‘ objects.

Author(s)

Russell Almond

See Also

MongoQueue

Examples

```
mq <- new("MongoQueue","QueueTest",mongo::MongoDB("Messages",noMongo=TRUE),
                  builder=buildMessage)
## Remove Fred's messages from the database.
cleanMessageQueue(mq,list(c(uid="Fred")),"QueueTest")
## Purge NO-OP messages from the imported data.
cleanMessageQueue(mq,list(c(mess="NO-OP")),"QueueTest")
```

`fetchNextMessage` *Returns the next unprocessed message from a message queue.*

Description

Searchers through the messages for the first unprocessed message. Return `NULL` if none is found.

Usage

```
fetchNextMessage(queue)
## S4 method for signature 'MessageQueue'
fetchNextMessage(queue)
```

Arguments

queue	The `MessageQueue` to search.

Details

The `ListQueue` message iterates through its internal collection until it finds an unprocessed message, or it runs out of messages. The `MongoQueue` message searches the collection.

Value

Either an object of class `P4Message` or `NULL` if there are no remaining unprocessed messages.

Note

The `ListQueue` method returns the current message if it has not been processed. Otherwise, it increments to pointer until if either finds an unprocessed messages or runs out of messages.

The `MongoQueue` method sorts the unprocessed messages by timestamp, and returns the one with the earliest message.

In both cases, `markAsProcessed` must be called on the processed message to advance the queue.

Author(s)

Russell Almond

See Also

`MessageQueue`, `markAsProcessed`, `resetProcessedMessages`

Examples

```
messy <- list(
  P4Message("test","Test 1","Tester","Test Message"),
  P4Message("test","Test 2","Tester","Test Message",processed=TRUE),
  P4Message("test","Test 3","Tester","Test Message"))
messq <- new("ListQueue","Qtest",messy)
mess1 <- fetchNextMessage(messq)
mess1
fetchNextMessage(messq)
markAsProcessed(messq,mess1)
mess2 <- fetchNextMessage(messq)
mess2
markAsProcessed(messq,mess2)
mess3 <- fetchNextMessage(messq)
```

---

generateListenerExports

Build tables from messages saved by the listner

Description

The function updateTable extracts a data table from the listener named by which, saves it into the named file. It then registers the generated file using registerOutput.

The function generateListenerExports calls the updateTable for each element in the export list, which should be a list of arguments to updateTable.

Usage

```
generateListenerExports(ls, exportlist, appid, outdir, process = ls$sender)
updateTable(ls, which, type, appid, outdir, fname = "<app>_<name>.csv",
            process = ls$sender, flattener = jsonlite::flatten, doc="",
            name=which)
```

Arguments

ls	The ListenerSet which contains both the listener and the registry.
exportlist	A list of lists of arguments to updateTable.
appid	A character scalar giving the name of the application. This should be the long name (e.g., "ecd://org/unit/assessment" not the short name ("assessment").
outdir	The path to the directory where the output should be stored.
process	A character scalare giving the name of the generating process. Passed to registerOutput.
which	An itentifier for which listener will generate the table, in other words, the name of one of the listeners.
type	A character string identifying the type of the output. Passed to registerOutput.

fname	A character vector giving a pattern for a file name. The string "<app>" is substituted for basename(app), the string "<name>" is substituted for name.
flattener	A function or string naming a function which is used to flatten nested data. See details.
name	Used to label the table in the registrity.
doc	A doc string added to the registrty.

Details

The updateTable function calls the listenerDataTable on the listener ls$listners[[which]]. As the details fields of the messages, could be nested, it might need to be flattened so that it can be exported as a CSV file, so the flattener function is called. Then the resulting data table is written out to outdir/fname.

The generateListenerExports is fed a list of arguments for updateTable. The idea is that this information can be included in the config.json file. Each element should be a list with the following components:

which Required, the name of the listener.

type Optional, the type of the output (for the registry); defaults to "data".

name Optional, the name of table in the registry. Defaults to which.

fname Optional, the file name. This is actually a pattern, and "<app>" is replaced with basename(appid) and "<name>" is replaced with name. Default is "<app>_<name>.csv".

flattener Optional, The name of the flatterner function. Defaults to flatten.

doc Optional, a character string describing the table in the registry.

Note that the appid, outdir and process fields are taken from the call to generateListenerExports.

Value

These functions are mainly used for their side effects. The updateTable function returns the generated table invisibly, or NULL if listenerDataTable returns NULL. The generateListenerExports returns the last exported table.

Flattening Complex Data

The data stored in the messages can in fact be nested deeply. So the raw dataframe returned by listenerDataTable could have columns that are themselves data frames. The function jsonlite::flatten function unrolls these columns into individual components.

Another frequenly used function is Peanut::flattenStats. In particular, the PnodeMargin statistic returns a labeled vector as output. This function splits it into columns with headers *name.state*. Note that to call a function from another package, that package must be named, so a call to require is in order.

Author(s)

Russell Almond

See Also

ListenerSet, Listener

listenerDataTable, registerOutput

flatten, flattenStats

Examples

```
## Not run:

  config.json <-
    '"listeners":[
 {"name":"ToAS",
  "type":"InjectionListener",
  "dbname":"ASRecords",
  "colname":"Statistics",
  "jsonEncoder":"unparseData",
  "jsonDecoder":"parseData",
  "messages":["Statistics"]
 },
 {"name":"PPStats",
  "type":"UpdateListener",
  "targetField":"data",
  "jsonEncoder":"stats2json",
  "colname":"Statistics",
  "messages":["Statistics"]
 }

    ],
    "listenerExports":[
 {"which":"PPStats",
  "type": "data",
  "fname":"stats-<app>.csv",
  "flattener":"flattenStats",
  "doc": "Reporting statistics"
        },
 {"which":"ToAS",
  "type": "hist",
  "fname":"hist-<app>.csv",
  "flattener":"flattenStats",
  "doc": "History of history variables."
        }
    ]'
  config <- jsonlite::fromJSON(config.json,FALSE)
  appid <- "ecd://example.edu/testgroup/test"
  outdir <- tempdir()
  ls <- buildListenerSet("EA",config$listeners, appid,
                         lscol="Messages",dbname="test",
                         dburi=mongo::makeDBuri(),
                         sslops=mongolite::ssl_options(),
                         registrycol="files",registrydbname="test")
  ## Need to make sure Peanut::flattenStats is recognized
```

```
  require(Peanut)

  updateTable(ls,"PPstats","data",appid,outdir)

  generateListenerExports(ls,config$listenerExports,appid,outdir)

## End(Not run)
```

`importMessages` *Imports a file full of messages into a message queue.*

Description

Interts the contents of a JSON file full of messages into the message queue.

Usage

```
importMessages(queue, filelist, data.dir)
## S4 method for signature 'MongoQueue'
importMessages(queue, filelist, data.dir)
```

Arguments

`queue`	An object of class `MessageQueue` to be loaded.
`filelist`	A list of filenames of files containing data to be loaded.
`data.dir`	A character scalar giving the pathname of the directory containing the data files.

Value

No particular return message. Used for its side effects.

Note

Current implementation uses the shell function ‘mongoimport‘ which may not be the best implementation if the Mongo server is on a different machine.

Author(s)

Russell Almond

See Also

`MongoQueue`, `cleanMessageQueue`

Examples

```
## Not run:
  mq <- MongoQueue("Test",mongo::MongoDB("TestMessages","test"))
 importMessages(mq,c("PretestResults.json","TestResults.json"),"/usr/local/share/Proc4/data/")

## End(Not run)
```

InjectionListener-class

Class "InjectionListener"

Description

This listener takes messages that match its incomming set and inject them into another Mongo database (presumably a queue for another service).

Details

The database is a mongo collection identified by dburi, dbname and colname (collection within the database). The mess field of the P4Message is checked against the applicable messages in messSet. If it is there, then the message is inserted into the collection.

Extends

This class implements the Listener interface.

All reference classes extend and inherit methods from "envRefClass".

Methods

isListener signature(x = "InjectionListener"): returns true.

receiveMessage signature(x = "InjectionListener", message): If the message is in the messSet, it saves the message to the database. (See details)

listenerName signature(x= "InjectionListener"): Returns the name assigned to the listener.

listenerDataTable signature(listener = "InjectionListener", appid): Builds a data datable from the messages.

Data Table

When the listenerDataTable method is called, a general find query (mdbFind on the backing collection. The app, uid, context, timestamp fields are selected, and the data (details) field is unpackaged and added as additional columns.

Fields

`sender`: Object of class `character` which is used as the sender field for the message.

`dbname`: Object of class `character` giving the name of the Mongo database

`dburi`: Object of class `character` giving the url of the Mongo database.

`colname`: Object of class `character` giving the column of the Mongo database.

`messSet`: A vector of class `character` giving the name of messages which are sent to the database. Only messages for which `mess(message)` is an element of `messSet` will be inserted.

`db`: Object of class `MongoDB` giving the database. Use `messdb()` to access this field to makes sure it has been set up.

Class-Based Methods

`messdb()`: Accessor for the database collection. Initializes the connection if it has not been set up.

`receiveMessage(message)`: Does the work of inserting the message. See Details.

`reset(app)`: Empties the database collection of messages with this app id.

`initialize(sender, dbname, dburi, colname, messSet, ...)`: Sets default values for fields.

Author(s)

Russell Almond

References

This is an example of the observer design pattern. https://en.wikipedia.org/wiki/Observer_pattern.

See Also

Listener, P4Message, InjectionListener, UpdateListener, UpsertListener, CaptureListener, TableListener, mongo

Examples

```
## Not run:

mess1 <- P4Message(app="default",uid="Phred",context="Down Hill",
                   sender="EIEvent",mess="New Observables",
                   details=list(trophy="gold",solvedtime=10))
ilwind <- InjectionListener(sender="EIEvent",messSet="New Observables")
receiveMessage(ilwind,mess1)


## End(Not run)
```

Listener *A listener is an object which can recieve a message.*

Description

A *listener* an an object that takes on the observer or listerner role in the the listener (or observer) design pattern. A listener will register itself with a speaker, and when the speaker sends a message it will act accordingly. The `receiveMessage` generic function must be implemented by a listener. It is called when the speaker wants to send a message.

Usage

```
receiveMessage(x, message)
isListener(x)
## S4 method for signature 'ANY'
isListener(x)
clearMessages(x, app)
listenerName(x)
listeningFor(x, newSet)
```

Arguments

`x`	A object of the virtual class `Listner`.
`message`	A `P4Message` which is being transmitted.
`app`	A character scalar identifying the application served by the listener.
`newSet`	A character vector giving the messages the listener is listening for. If empty, the listener processes all messages it recieves.

Details

The `RefListener` class is an abstract class. Any object can become a listener by giving it a method for `receiveMessage`. The message is intended to be a subclass of `P4Message`, but in practice, no restriction is placed on the type of the message.

As `RefListener` an abstract class, it means definition. Instead the generic function `isListner` is used to test if the object is a proper listener or not. The default method checks for the presence of a `receiveMessage` method. As this might not work properly with S3 objects, an object can also register itself directly by setting a method for `isListner` which returns true.

Typically, a lister will register itself with the speaker objects. For example the `ListenerSet$addListener` method adds itself to a list of listeners maintained by the object. When the `ListenerSet$notifyListeners` method is called, the `receiveMessage` method is called on each listener in the list.

Value

The `isListener` function should return `TRUE` or `FALSE`, according to whether or not the object follows the listner protocol.

The `listenerName` returns a character scalar with the name of the listener.

The receiveMessage and clearMessages functions are typically invoked for side effects and it may have any return value.

The listeningFor function returns a character vector giving the messages used by the listener.

Fields

name An identifier for the listener, mainly used in error messages.

messSet A character vector giving the messages list listener will process. Messages whose mess field are not in the list are not processed. As a special case, if messSet has length 0, then all messages are processed.

db An object of class MongoDB, which contains a database to contain the messages. Note: if the subclass does not use this, then the connection to the database will not be made.

Methods

isListener signature(x = "RefListener"): Returns true, as subclasses of RefListener follow the listener protocol.

receiveMessage signature(x = "RefListener",message="P4Message"): This first checks to see if mess{(message)} is in the messList field. If so, it delegates the processing of the message to the $recieveMessage() method. This class-based method must be implemented in subclasses.

clearMessages signature(sender = "RefListener", app="character"): This delegates the process of cleaning the message collection to the $reset() class method.

listenerName signature(x="RefListener"): Returns the name of the listener.

listeningFor signature(x="RefListener", newSet="chracter"): Returns the names of the messages this listener is listening for. If newSet is supplied, the message set is updated.

Class-Based Methods

initialize(name,db,messSet,...): Provides default values for various fields.

messdb(): Returns the MongoDB object in the db field.

receiveMessage(message): Does the message processing. Note that the RefListener method returns and error, so *subclasses must implement this*. Note, also that the filtering of which messages to handle is done by the S4 method.

reset(app): This method clears out the old messages. Again, *subclasses must implement this method* as the RefListener class implementation raises an error. The app argument is because several different implementations may store messages for more than one application in the same place.

listeningFor(newSet): This method returns, or if the second argument in present, sets the messSet field.

Author(s)

Russell Almond

References

https://en.wikipedia.org/wiki/Observer_pattern

See Also

Implementing Classes: CaptureListener, UpdateListener, UpsertListener, InjectionListener, TableListener

Related Classes: ListenerSet, P4Message

Examples

```
setRefClass("FileListener",fields=c(file="character"),
            contains="RefListener",
            methods=c(
                receiveMessage = function (message) {
                  cat("I (",listenerName(.self),
                      ") just got the message ",
                      mess(message),
                      file=file,append=TRUE)
                },
                reset = function(app) {
                  cat("\f",file=file,append=TRUE)
                }
            ))

myListener <- new("FileListener",name="Test",file="",
                  messSet="Scored Response",
                  db=mongo::MongoDB(noMongo=TRUE))

mess1 <- P4Message("Fred","Task 1","Evidence ID","Scored Response",
         as.POSIXct("2018-11-04 21:15:25 EST"),
         list(correct=TRUE,seletion="D"))

mess2 <- P4Message("Fred","Task 2","Evidence ID","Raw Response",
         as.POSIXct("2018-11-04 21:16:45 EST"),
         list(seletion="C"))

isListener(myListener)
listenerName(myListener)
receiveMessage(myListener,mess1) ## This one is processed.
receiveMessage(myListener,mess2) ## This one is ignored.
clearMessages(myListener,"")
```

ListenerConstructors *Constructors for Listener Classes*

Description

These functions create objects of class CaptureListener, UpdateListener, UpsertListener, InjectionListener, and TableListener.

Usage

```
CaptureListener(name="Capture",messages = list(),
                messSet=character(), ...)
InjectionListener(name="Injection", db = mongo::MongoDB(noMongo=TRUE),
    messSet = character(), ...)
UpdateListener(name="Update",db = mongo::MongoDB(noMongo=TRUE),
               targetField = "data", qfields = c("app", "uid"),
               jsonEncoder = "unparseData", jsonDecoder="parseData",
               messSet=character(), ...)
UpsertListener(name="Upsert", messSet = character(),
               db = mongo::MongoDB(noMongo=TRUE),
               qfields = c("app", "uid"), ...)
TableListener(name = "ppData",
    fieldlist = c(uid = "character", context = "character"),
    messSet = character(), ...)
```

Arguments

messages	A list into which to add the messages.
messSet	A character vector giving the message values of the messages that will be processed. Messages whose mess value are not in this list will be ignored by this listener.
db	A MongoDB object which provides reference to a database collection.
targetField	The name of the field that will be modified in the database by the UpdateListener.
jsonEncoder	A function that will be used to encode the data object as JSON before it is set. See UpdateListener.
jsonDecoder	A function that will be used to decode the data object from JSON when building tables. See UpdateListener.
qfields	The fields that will be used as a key when trying to find matching messages in the database for the UpsertListener.
name	An object of class character naming the listener.
fieldlist	A named character vector giving the names and types of the columns of the output matrix. See TableListener.
...	Other arguments passed to the constructor.

Details

The functions are as follows:

CaptureListener Creates an object of class CaptureListener which stores the messages in a list.

InjectionListener Creates an object of class InjectionListener which inserts the message into the designated database.

UpdateListener Creates an object of class UpdateListener which updates the designated field.

`UpsertListener` Creates an object of class `UpsertListener` which insert or replaces the message in the designated collection.

`TableListener` Creates an object of class `TableListener` which adds details from message to rows of a data frame.

See the class descriptions for more information.

Value

An object of the virtual class `Listener`.

Author(s)

Russell Almond

References

This is an example of the observer design pattern. https://en.wikipedia.org/wiki/Observer_pattern.

See Also

`Listener`, `P4Message`, `UpsertListener`, `UpdateListener`, `CaptureListener`, `InjectionListener`, `TableListener`, `ListenerSet`, `mongo`

Examples

```
cl <- CaptureListener()

il <- InjectionListener("Evidence Collector",
                        db=mongo::MongoDB(collection="EvidenceSets",
                                          db="EARecords",
                                          url = "mongodb://localhost",
                                          noMongo=TRUE),
                        messSet="New Observables")

upsl <- UpsertListener("Save Observables",
                        db=mongo::MongoDB(collection="LatestEvidence",
                                          db="EARecords",
                                          url = "mongodb://localhost",
                                          noMongo=TRUE),
                       messSet="New Observables", qfields=c("app","uid"))

trophy2json <- function(dat) {
  paste('{', '"trophyHall"', ':','[',
        paste(
            paste('{"',names(dat$trophyHall),'":"',dat$trophyHall,'"}',
                  sep=""), collapse=", "), '],',
        '"bankBalance"', ':', dat$bankBalance, '}')
}
ul <- UpdateListener("Player Data",
                     db=mongo::MongoDB(collection="Players",
```

```
                                        db="Proc4",
                                        url = "mongodb://localhost",
                                        noMongo=TRUE),
                        targetField="data",
                        messSet=c("Money Earned","Money Spent"),
                        jsonEncoder="trophy2json")

tabMaker <- TableListener(name="Trophy Table",
                          messSet="New Observables",
                          fieldlist=c(uid="character", context="character",
                                      timestamp="character",
                                      solvedtime="numeric",
                                      trophy="ordered(none,silver,gold)"))
```

`listenerDataTable` *Fetches a data frame containing information captured by listener*

Description

A number of listerner capture information. This method extracts the data as a data frame for further processing.

Usage

```
listenerDataTable(listener, appid = character())
## S4 method for signature 'RefListener'
listenerDataTable(listener, appid = character())
```

Arguments

`listener`	A `Listener` subclass where the information was stored.
`appid`	The name of the application whose data is to be extracted. (In case data from more than one application is stored in the same collection.)

Value

A `data.frame` giving the requested data.

Note that this data frame could in fact contain columns which are themselves data frames. Consider calling `jsonlite::flatten` or `Peanut::flattenStats` on the output.

Author(s)

Russell Almond

See Also

`InjectionListener`, `registerOutput`

Examples

```
## Not run:
jspecs <- '{
     "listeners":[
 {
     "name":"ToAS",
     "type":"InjectionListener",
     "dbname":"ASRecords",
     "colname":"Statistics",
     "messages":["Statistics"]
 }

    ]}'

speclist <- jsonlite::fromJSON(jspecs,FALSE)
appid <- "ecd://pluto.coe.fsu.edu/P4Test"
outdir <- "/usr/local/share/Proc4/data"

lset <- buildListenerSet("TestEngine",speclist$listeners,
                         appid=appid,
                         lscol="Messages",dbname="test",
                         dburi="", sslops=mongolite::ssl_options(),
                         registrycol="OutputFiles",
                         registrydbname="Proc4")


## After engine running.

sl <- lset$listeners[["ToAS"]]
sdat <- listenerDataTable(sl,NULL,appid)
registerOutput(ls,"PP Statistics",
               file.path(outdir,"PPstats.csv"),
               appid,"EA")


## End(Not run)
```

`ListenerSet-class` *Class* `"ListenerSet"`

Description

This is a "mix-in" class that adds a speaker protocol to an object, which is complementary to the `Listener` protocol. This object maintains a list of listeners. When the `notifyListeners` method is called, it notifies each of the listeners by calling the `receiveMessage` method on the listener.

Extends

All reference classes extend and inherit methods from `"envRefClass"`. The class union `NullListenerSet` is either a `ListenerSet` or `NULL`.

Methods

isListener `signature(x = "ListenerSet")`: Returns true, as the ListenerSet follows the listener protocol.

receiveMessage `signature(x = "ListenerSet")`: A synonym for `notifyListeners`.

notifyListeners `signature(sender = "ListenerSet")`: A synonym for the `notifyListeners` internal method.

Protocol

The key to this class is the `notifyListeners` method. This method should receive as its argument a `P4Message` object. (The protocol is fairly robust to the type of message and the type is not enforced. In fact, any object which has a `as.jlist` method should work.)

When the notifier is called it performs the following functions:

1. It saves the message to the collection represented by `messdb()`. If `messdb()` is `NULL` (`dburi` is the empty string) then the messages is not saved.
2. It calls the `receiveMessage` method on each of the objects in the listener list.
3. It logs the messages sent using the `flog.logger`, in the `"Proc4"` logger. The sending of the messages is logged a the "INFO" level, and the actual message at the "DEBUG" level.

In addition, the `ListenerSet` maintains a named list of `Listener` objects (that is, objects that have a `receiveMessage` method). The methods `addListener` and `removeListener` maintain this list.

Fields

`sender`**:** Object of class `character`:the name of the source of the messages.

`dburi`**:** Object of class `character`: the URI for the `mongo` database. If null, then no recording of messages to a database is done (except possibly in the listeners).

`dbname`**:** Object of class `character`: the name of the database in which messages should be logged.

`colname`**:** Object of class `character`: the name of the collection in which messages should be logged.

`listeners`**:** A named `list` of `Listener` objects, that is objects for which `isListener` is true.

`db`**:** Object of class `MongoDB` which is a handle to the collection where messages are logged, or `NULL` if the log database has not been initialized. As the database may have not been initialized, programs should call the `messdb()` method which will open the database connection if it is not yet open.

Class-Based Methods

`$notifyListeners(mess)`**:** This method calls `receiveMessage` on all of the listeners. See Protocol section above.

`$addListener(name, listener)`**:** This method addes a lsitener to the list.

`$initialize(sender, dburi, listeners, colname, ...)`**:** This creates the listener. Note, this does not initialize the database collection. Call `messdb()` to initialize the collection.

`$removeListener(name)`**:** This removes a listener from the collection by its name.

`$reset(app)`: Empties the database collection of messages with this app id.

$messdb `signature()`: Returns the `mongo` database collection to which to log messages. Creates the column if it has not been initialized.

$registrydb `signature()`: Returns the `mongo` database collection in which output files will be registered.

$registerOutput `signature(name, filename, app, process, type="data", doc="")`: Adds/updates a field in the database collection of output files. This allows processes looking at the database to find output summaries.

Note

The `notifyListeners` method uses the `flog.logger` protocol. In particular, it logs sending the message at the "INFO" level, and the actual message sent at the "DEBUG" level. In particular, setting `flog.threshold(DEBUG,name="Proc4")` will turn on logging of the actual message and `flog.threshold(WARN,name="Proc4")` will turn off logging of the message sent messages.

It is often useful to redirect the Proc4 logger to a log file. In addition, changing the logging format to JSON, will allow the message to be recovered. Thus, try `flog.layout(layout.json,name="Proc4"` to activate logging in JSON format.

Author(s)

Russell Almond

References

https://en.wikipedia.org/wiki/Observer_pattern

See Also

`Listener`, `receiveMessage`, `notifyListeners`, `flog.logger`, `mongo`, `P4Message`

Listener Classes. `CaptureListener`, `UpdateListener`, `UpsertListener`, `InjectionListener`, `TableListener`

Examples

```
showClass("ListenerSet")
```

ListQueue-class *Class* "ListQueue"

Description

This is a minimal implementation of the `MessageQueue` abstract class. In this case, the messages are just held in an internal array. It probably works well for short queues, and does not require a database or other external connection, so it useful for testing.

Details

The queue is implemented with a list and a pointer to the current position in the list. The `$hasNext()` and `$nextMessage()` methods implement a typical iterator paradigm.

Note that the `MessageQueue` paradigm is slightly different. Here the current message is the one returned by `fetchNextMessage()` until is is marked as processed (`markAsProcessed()`), which will then cause the `$nextMessage()` method to be called advancing the postition.

Extends

Class `"MessageQueue"`, directly.

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

fetchNextMessage `signature(queue = "MessageQueue")`: Returns the next unprocessed message from the Queue, or 'NULL' if there are no processed messages in the queue.

markAsError `signature(col = "ListQueue", mess = "ANY")`: Marks a message as an error, and saves error message in queue.

markAsProcessed `signature(col = "ListQueue", mess = "ANY")`: Marks a message as processed.

resetProcessedMessages `signature(queue = "MongoQueue")`: Clears the processed flag for messages matching the query.

Note, that currently there is no implementing method for `cleanMessageQueue` or `importMessages`.

Fields

`app`: Object of class `character` giving ID of application.

`messages`: Object of class `list` giving the messages.

`pos`: Object of class `integer` giving the current position of the queue.

Class-Based Methods

`$nextMessage()`: Advances the position, and returns the next message (or 'NULL' if all messages have been returned.

`$getCurrent()`: Returns the current message.

`$initialize(app, messages, ...)`: initializer

`$reset()`: Resets the position to the beginnnig of the queue.

`$setCurrent(newmess)`: Updates the message at the current position.

`hasNext()`: Returns true if there are more messages in the queue.

`fetchNextMessage()`: Fetches the next unprocessed message. This is either the current message, if not processed, or the `$nextMessage()` method is called until the first unprocessed message is found.

`$count()`: Returns the number of messages remaining in queue. Note, count includes both processed and unprocessed messages.

Note

This is an experimental implementation, and details may change in future release.

Author(s)

Russell Almond

See Also

MessageQueue,P4Message, MongoDB, fetchNextMessage(), cleanMessageQueue(), importMessages(), markAsProcessed(), resetProcessedMessages(), buildMessage(), getOneRec()

Examples

```
showClass("ListQueue")
```

---

markAsProcessed *Functions for manipulating entries in a message queue.*

Description

A collection of message objects can serve as a queue: they can be sorted by their timestamp and then processed one at a time. The function markAsProcessed sets the processed flag on the message and then saves it back to the database. The function processed returns the processed flag.

The function markAsError attaches an error to the message and saves it. The function processingError returns the error (if it exists).

Usage

```
markAsProcessed(col, mess)
## S4 method for signature 'JSONDB,P4Message'
markAsProcessed(col, mess)
## S4 method for signature 'ListQueue'
markAsProcessed(col, mess)
## S4 method for signature 'MongoQueue,ANY'
markAsProcessed(col, mess)
## S4 method for signature 'NULL,P4Message'
markAsProcessed(col, mess)
markAsError(col, mess, e)
## S4 method for signature 'JSONDB,P4Message'
markAsError(col, mess, e)
## S4 method for signature 'ListQueue'
markAsError(col, mess, e)
## S4 method for signature 'MongoQueue,ANY'
markAsError(col, mess, e)
## S4 method for signature 'NULL,P4Message'
markAsError(col, mess, e)
```

```
processed(x)
processingError(x)
```

Arguments

mess	An object of class P4Message to be modified.
col	A MongoDB collection where the message queue is stored (or an object which wraps sucha collection). This can also be NULL in which case the message will not be saved to the database.
e	An object indicating the error occurred. Note this could be either a string giving the error message of an object of an error class. In either case, it is converted to a string before saving.
x	A message object to be queried.

Details

A MongoDB collection of messages can serve as a queue (see MongoQueue). As messages are added into the queue, the processed flag is set to false. The handler then fetches them one at a time (sorting by the timestamp). It then does whatever action is required to handle the message. Then the function markAsProcessed is called to set the processed flag to true and update the entry in the database.

Some thought needs to be given as to how to handle errors. The function markAsError attaches an error object to the message and then updates it in the collection. The error object is turned into a string (using toString) before saving, so it can be any type of R object (in particular, it could be either the error message or the actual error object thrown by the function).

Value

The functions markAsProcessed and markAsError both return the modified message.

The function processed returns a logical value indicating whether or not the message has been processed.

The function processingError returns the error object attached to the message, or NULL if no error object is returned. Note that the error object could be of any type.

Note

The functions markAsProcessed and markAsError do not save the complete record, they just update the processed or error field.

There was a bug in early version of this function, which caused the error to be put into a list when it was saved. This needs to be carefully checked.

Author(s)

Russell Almond

See Also

P4Message, getOneRec, buildJQuery, timestamp, MessageQueue, resetProcessedMessages

Examples

```
## Not run:
col <- mongolite::mongo("TestMessages")
col$remove('{}')            # Clear out anything else in queue.
mess1 <- P4Message("One","Adder","Tester","Add me",app="adder",
                   details=list(x=1,y=1))
mess2 <- P4Message("Two","Adder","Tester","Add me",app="adder",
                   details=list(x="two",y=2))
mess1 <- saveRec(mess1,col,FALSE)
mess2 <- saveRec(mess2,col,FALSE)

mess <- getOneRec(buildJQuery(app="adder", processed=FALSE),
    col, parseMessage, sort = c(timestamp = 1))
iterations <- 0
while (!is.null(mess)) {
  if (iterations > 4L)
    stop ("Test not terminating, flag not being set?")
  iterations <- iterations + 1
  print(mess)
  print(details(mess))
  out <- try(print(details(mess)$x+details(mess)$y))
  if (is(out,'try-error'))
   mess <- markAsError(mess,col,out)
  mess <- markAsProcessed(mess,col)
  mess <- getOneRec(buildJQuery(app="adder", processed=FALSE),
    col, parseMessage, sort = c(timestamp = 1))

}

mess1a <- getOneRec(buildJQuery(app="adder",uid="One"),col,parseMessage)
mess2a <- getOneRec(buildJQuery(app="adder",uid="Two"),col,parseMessage)
stopifnot(processed(mess1a),processed(mess2a),
          is.null(processingError(mess1a)),
          grepl("Error",processingError(mess2a)))

## End(Not run)
```

MessageQueue-class *Class* "MessageQueue"

Description

A message queue is an ordered collection of P4Message objects. The principle idea is that the fetchNextMessage() function will fetch the next unprocessed message, and consequently, this can be used to schedule the work for a scoring engine.

Details

The general queue functions are determined by two generic functions: fetchNextMessage(), and markAsProcessed(). The fetchNextMessage() returns the "first" (the meaning of first is defined

by the implementing Queue object) unprocssed message (i.e., `processed(mess)=FALSE`). Note that the `fetchNextMessage()` function will continue to return the same message until it is marked as processed using `markAsProcessed(queue,mess)`. Note that simply setting `processed(mess) <- FALSE` is not sufficient because the change is not stored in the queue.

Extends

All reference classes extend and inherit methods from `"envRefClass"`.

Fields

`app`: Object of class `character` giving the name of the application.

GenericFunctions

The following generic functions are designed to work with subclasses of message queues, however, currently only the `MongoQueue` has all of the methods.

cleanMessageQueue: Removes messages matching query from queue.

fetchNextMessage: Returns the next unprocessed message from the Queue, or 'NULL' if there are no processed messages in the queue.

importMessages: Imports messages into a queue from a file.

markAsError: Marks a message as an error, and saves error message in queue.

markAsProcessed: Marks a message as processed.

resetProcessedMessages: Clears the processed flag for messages matching the query.

Class-Based Methods

`$initialize(app, ...)`: Constructor.

`$count()`: Returns the number of messages remaining in queue.

Note

The current implementations are `MongoQueue` which uses a database collection for the queue, and a partially implemented `ListQueue` which just uses an array of messages. This is not fully implemented.

Some other alternatives would be to link to a formal queuing system, like Kafka, or to some kind of RPC server.

Author(s)

Russell Almond

See Also

`MongoQueue`, `ListQueue`, `P4Message`, `MongoDB`, `fetchNextMessage()`, `cleanMessageQueue()`, `importMessages()`, `markAsProcessed()`, `resetProcessedMessages()`, `buildMessage()`, `getOneRec()`

Examples

```
showClass("MessageQueue")
```

---

mongoAppender-class *Class* "mongoAppender"

Description

This implements the appender protocol logging to a database. Note that flog.appender expects a function as its argument. The $logger() method returns a function which can be passed to flog.appender.

Extends

All reference classes extend and inherit methods from "envRefClass".

Fields

db: Object of class JSONDB the refernce to the column where the log will be stored.

app: Object of class character The application identifier for which we are logging errors. (See app()).

engine: Object of class character giving the name of the processes (in the 4 Process sense) that is generating the messages.

tee: Object of class character if this has length greater than zero, it should be a file to which the log is also sent. If it is "", then the log message is sent to standard output.

Methods

logit(line): This does the work of logging a line.

logger(): This returns a function which does the logging.

Author(s)

Russell Almond

See Also

flog.appender MongoDB

Examples

```
col <- mongo::MongoDB("ErrorLog","Admin",noMongo=TRUE)
logfile <- tempfile("testlog","/tmp",fileext=".log")
apnd <- mongoAppender(db=col,app="p4test",engine="Tester",tee=logfile)
futile.logger::flog.appender(apnd$logger(),"TEST")
```

MongoQueue-class *Class* "MongoQueue"

Description

This is a message queue implemented as a database collection.

This wraps a collection in a Mongo (or other JSON-based) database. The `fetchNextMessage` looks for the first (earliest timestamp) message which is not processed.

Extends

Class "`MessageQueue`", directly.

All reference classes extend and inherit methods from "`envRefClass`".

Methods

cleanMessageQueue `signature(queue = "MongoQueue")`: Removes messages matching query from queue.

fetchNextMessage `signature(queue = "MessageQueue")`: Returns the next unprocessed message from the Queue, or 'NULL' if there are no processed messages in the queue.

importMessages `signature(queue = "MongoQueue")`: Imports messages into a queue from a file.

markAsError `signature(col = "MongoQueue", mess = "ANY")`: Marks a message as an error, and saves error message in queue.

markAsProcessed `signature(col = "MongoQueue", mess = "ANY")`: Marks a message as processed.

resetProcessedMessages `signature(queue = "MongoQueue")`: Clears the processed flag for messages matching the query.

Fields

`app`: Object of class `character` giving the identifier of the application. This is used to restrict the `app` field of the message to match the current application.

`messDB`: Object of class `JSONDB` that provides a reference to the database collection storing the messages.

`builder`: Object of class `function` which is used to reconstruct the messages from the data, see `buildMessage()` and `getOneRec()`.

Class-Based Methods

`$queue()`: Returns the `JSONDB` reference.

`$initialize(app, messDB, builder, ...)`: Constructor.

`$fetchNextMessage()`: Internal implementation of the fetch method.

`$buildIndex()`: This method builds an index in the collection. Generally only needs to be done once.

`$count()`: Returns the number of unprocessed messages remaining in queue.

Note

It is probably a good idea to build an index on this database using the "processed" and "timestamp" fields. The `$buildIndex()` method does this.

Author(s)

Russell Almond

See Also

`MessageQueue`,`P4Message`, `MongoDB`, `fetchNextMessage()`, `cleanMessageQueue()`, `importMessages()`, `markAsProcessed()`, `resetProcessedMessages()`, `buildMessage()`, `getOneRec()`

Examples

```
showClass("MongoQueue")
```

---

notifyListeners *Notifies listeners that a new message is available.*

Description

This is a generic function for objects that send `P4Message` objects. When this function is called, the message is sent to the listeners; that is, the `receiveMessage` function is called on the listener objects. Often, this protocol is implemented by having the `sender` include a `ListenerSet` object.

Usage

```
notifyListeners(sender, message)
```

Arguments

`sender`	An object which sends messages.
`message`	A `P4Message` to be sent.

Value

Function is invoked for its side effect, so return value may be anything.

Author(s)

Russell Almond

See Also

`P4Message`, `Listener`, `ListenerSet`

Examples

```
## Not run: ## Requires Mongo database set up.
MyListener <- setClass("MyListener",slots=c("name"="character"))
setMethod("receiveMessage","MyListener",
   function(x,mess)
      cat("I (",x@name,") just got the message ",mess(mess),"\n"))


lset <-
ListenerSet$new(sender="Other",dburi="mongodb://localhost",
                colname="messages")
lset$addListener("me",MyListener())

mess1 <- P4Message("Fred","Task 1","Evidence ID","Scored Response",
         as.POSIXct("2018-11-04 21:15:25 EST"),
         list(correct=TRUE,seletion="D"))

mess2 <- P4Message("Fred","Task 2","Evidence ID","Scored Response",
         as.POSIXct("2018-11-04 21:17:25 EST"),
         list(correct=FALSE,seletion="D"))

lset$notifyListeners(mess1)

lset$removeListener("me")

notifyListeners(lset,mess2)


## End(Not run)
```

P4Message *Constructor and accessors for P4 Messages*

Description

The function P4Message() creates an object of class "P4Message". The other functions access fields of the messages.

Usage

```
P4Message(uid, context, sender, mess, timestamp = Sys.time(),
          details = list(), app = "default", processed=FALSE)
app(x)
app(x) <- value
uid(x)
uid(x) <- value
mess(x)
mess(x) <- value
context(x)
```

```
context(x) <- value
sender(x)
sender(x) <- value
timestamp(x)
timestamp(x) <- value
details(x)
details(x) <- value
## S4 method for signature 'P4Message'
toString(x,...)
## S4 method for signature 'P4Message'
show(object)
## S3 method for class 'P4Message'
all.equal(target, current, ..., checkTimestamp = FALSE,
                                check_ids = TRUE)
```

Arguments

uid	A character object giving an identifier for the user or student.
context	A character object giving an identifier for the context, task, or item.
sender	A character object giving an identifier for the sender. In the four-process architecture, this should be one of "Activity Selection Process", "Presentation Process", "Evidnece Identification Process", or "Evidence Accumulation Process".
mess	A character object giving a message to be sent.
timestamp	The time the message was sent.
details	A list giving the data to be sent with the message.
app	An identifier for the application using the message.
processed	A logical flag: true if the message has been processed and false otherwise.
x	A message object to be queried, or converted to a string.
...	Addtional arguments for show or all.equal.
object	A message object to be converted to a string.
target	A P4Message to compare.
current	A P4Message to compare.
checkTimestamp	Logical flag. If true, the timestamps are compared as part of the equality test.
check_ids	Logical flag. If true, the database ids are compared as part of the equality test.
value	A new value for the field, type varies, but usually character.

Details

This class represents a semi-structured data object with certain header fields which can be indexed plus the free-form `details()` field which contains the body of the message. It can be serielized in JSON format (using `as.json` in the Mongo database (using the `mongolite` package).

Using the public methods, the fields can be read but not set. The generic functions are exported so that other object can extend the `P4Message` class. The `m_id` function accesses the mongo ID of the object (the `_id` field).

The function `all.equal.P4Message` checks two messages for identical contents. The flags `checkTimestamp` and `check_ids` can be used to suppress the checking of those fields. If timestamps are checked, they must be within .1 seconds to be considered equal.

Value

An object of class `P4Message`.

The `app()`, `uid()`, `context()`, `sender()`, and `mess()` functions all return a character scalar. The `timestamp()`, function returns an object of type `POSIXt` and the `details()` function returns a list.

The function `all.equal.P4Message` returns either `TRUE` or a vector of mode "character" describing the differences between `target` and `current`.

Author(s)

Russell G. Almond

References

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

See Also

`P4Message` — class `buildMessage`, `saveRec`, `getOneRec`

Examples

```
mess1 <- P4Message("Fred","Task 1","Evidence ID","Scored Response",
         as.POSIXct("2018-11-04 21:15:25 EST"),
         list(correct=TRUE,selection="D"))
stopifnot(
  app(mess1) == "default",
  uid(mess1) == "Fred",
  context(mess1) == "Task 1",
  sender(mess1) == "Evidence ID",
  mess(mess1) == "Scored Response",
  timestamp(mess1) == as.POSIXct("2018-11-04 21:15:25 EST"),
  details(mess1)$correct==TRUE,
  details(mess1)$selection=="D"
)

mess2 <- P4Message("Fred","Task 1","Evidence ID","Scored Response",
         as.POSIXct("2018-11-04 21:15:25 EST"),
         list(correct=FALSE,selection="E"))
all.equal(mess1,mess2)
stopifnot(!isTRUE(all.equal(mess1,mess2)))
```

P4Message-class *Class* "P4Message"

Description

This is a message which is sent from one process to another in the four process architecture. There are certain header fields with are used to route the message and the details field which is an arbitrary list of data which will can be used by the receiver.

This class represents a semi-structured data object with certain header fields which can be indexed plus the free-form `details()` field which contains the body of the message. It can be serielized in JSON format (using `as.json`) or saved in the Mongo database (using the `mongolite` package).

Objects from the Class

Objects can be created by calls to the `P4Message()` function.

Message Queues

Because all messages have a processed flag and a timestamp, a message collection becomes a queue. Simply search for the message with the earliest timestamp with `processed(mess)==FALSE` and excute that. Then sets processed equal to true using `markAsProcessed`.

If an error occurs during processing, the error can be associated with the message by setting the pError field using `markAsError`.

Slots

`_id`: Used for internal database ID.

`app`: Object of class `"character"` which specifies the application in which the messages exit.

`uid`: Object of class `"character"` which identifies the user (student).

`context`: Object of class `"character"` which identifies the context, task, or item.

`sender`: Object of class `"character"` which identifies the sender. This is usually one of "Presentation Process", "Evidence Identification Process", "Evidence Accumulation Process", or "Activity Selection Process".

`mess`: Object of class `"character"` a general title for the message context.

`timestamp`: Object of class `"POSIXt"` which gives the time at which the message was generated.

`data`: Object of class `"list"` which contains the data to be transmitted with the message.

`processed`: A logical value: true if the message has been processed, and false if the message is still in queue to be processed. This field is set with `markAsProcessed`.

`pError`: If a error occurs while processing this event, information about the error can be stored here, either as an R object, or as an R object of class error (or any class). This field is accessed with `processingError` and set with `markAsError`.

Methods

m_id `signature(x = "ANY")`: returns the `_id` field, the database ID.

app `signature(x = "P4Message")`: returns the app field.

as.jlist `signature(obj = "P4Message", ml = "list")`: coerces the object into a list to be processed by `toJSON`.

as.json `signature(x = "P4Message")`: Coerces the message into a JSON string.

context `signature(x = "P4Message")`: returns the context field.

details `signature(x = "P4Message")`: returns the data associated with the message as a list.

mess `signature(x = "P4Message")`: returns the message field.

sender `signature(x = "P4Message")`: returns the sender field.

timestamp `signature(x = "P4Message")`: returns the timestamp.

uid `signature(x = "P4Message")`: returns the user ID.

processing `signature(x = "P4Message")`: returns a logical value indicated whether or not the message has been marked as processed.

processingError `signature(x = "P4Message")`: if an error occurred while processing this message, returns a value describing the error. Otherwise, returns NULL.

Author(s)

Russell G. Almond

References

Almond, R. G., Steinberg, L. S., and Mislevy, R.J. (2002). Enhancing the design and delivery of Assessment Systems: A Four-Process Architecture. *Journal of Technology, Learning, and Assessment*, **1**, http://ejournals.bc.edu/ojs/index.php/jtla/article/view/1671.

See Also

`P4Message()` — constructor `buildMessage`, `saveRec`, `getOneRec`

Examples

```
showClass("P4Message")
```

registerOutput *Registers a file used for output information from an engine.*

Description

Many scoring engines provide output data files, or log file. This function registers the output files in a database collection, so that other functions can find them.

Usage

```
registerOutput(registrar, name, filename, app, process, type = "data", doc = "")
```

Arguments

registrar	The ListenerSet responsible for registering the the file.
name	A character scalar identifying the data file.
filename	A character scalar giving the path to the file.
app	A character scalar identifying the application
process	A character scalar identifying the name of the process (engine) generating the data).
type	A character scalar identify the data type. Currently supported values are "data" for data files in csv format, and "log" for log files.
doc	An object of type character describing the file.

Details

The file `system.file("dongle/Status.php",package="Proc4")` provides a web interface listing the output files. It generates this by looking at the "OutputFile" collection in the "Proc4" database. It then builds links to the files, so they can be downloaded.

The `registerOutput` method is used to add, or update the date on files in the collection.

Value

Mostly used for is side-effects. Returns information about the success of the database operation.

Author(s)

Russell Almond

See Also

ListenerSet, listenerDataTable

Examples

```
## Not run:
jspecs <- '{
     "listeners":[
 {
     "name":"ToAS",
     "type":"InjectionListener",
     "dbname":"ASRecords",
     "colname":"Statistics",
     "messages":["Statistics"]
 }

    ]}'

speclist <- jsonlite::fromJSON(jspecs,FALSE)
appid <- "ecd://pluto.coe.fsu.edu/P4Test"
outdir <- "/usr/local/share/Proc4/data"

lset <- buildListenerSet("TestEngine",speclist$listeners,
                         appid=appid,
                         lscol="Messages",dbname="test",
                         dburi="", sslops=mongolite::ssl_options(),
                         registrycol="OutputFiles",
                         registrydbname="Proc4")


## After engine running.

sl <- lset$listeners[["ToAS"]]
sdat <- listenerDataTable(sl,NULL,appid)
registerOutput(ls,"PP Statistics",
               file.path(outdir,"PPstats.csv"),
               appid,"EA")


## End(Not run)
```

`resetListeners` *Clears messages caches associated with listeners*

Description

Listeners often cache the messages in some way. This causes the message cache to be cleared, and operation which is often useful before a rerun. The `which` argument is used to control which listeners should have their cache cleared.

Usage

```
resetListeners(x, which, app)
```

```
## S4 method for signature 'ListenerSet'
resetListeners(x, which, app)
## S4 method for signature 'NULL'
resetListeners(x, which, app)
```

Arguments

x	A ListenerSet object containing the listeners to be reset.
which	A character vector containing the names of the listeners to reset. The special keyword "ALL" means all listeners will be reset. The special keyword "Self" means that the cache associated with the listener set will be reset.
app	A global applicaiton identifier. The reset operation should only be applied to messages from this application.

Details

Each Listener object (including the listener set) has a $reset() method which empties the cache of messages. This method calls the $reset() method for each of the listeners named in which. The special keyword "ALL" is used to reset all listeners and the special keyword "Self" is used to refer to the ListenerSet object itself (which may have a database colleciton).

Value

The ListenerSet object is returned.

Author(s)

Russell Almond

See Also

ListenerSet, Listener

Examples

```
## Not run: ## Requires Mongo database set up.

data2json <- function(dat) {
  toJSON(sapply(dat,unboxer))
}

listeners <- list(
  cl = CaptureListener(name="cl"),
  upd = UpdateListener(name="upd",messSet="New Observables",
                       dburi="mongodb://localhost",dbname="test",
                       targetField="data",jsonEncode="data2json",
                       colname="Updated"),
  ups = UpsertListener(name="ups",sender="EIEvent",messSet="New Observables",
                       dburi="mongodb://localhost",dbname="test",
                       colname="Upserted", qfields=c("app","uid")),
  il = InjectionListener(name="il",sender="EIEvent",messSet="New Observables",
```

```
                        dburi="mongodb://localhost",dbname="test",
                        colname="Injected"),
  tl = TableListener(name="tl",
                  messSet="New Observables",
                  fieldlist=c(uid="character", context="character",
                              timestamp="character",
                              solvedtime="numeric",
                              trophy="ordered(none,silver,gold)"))
  )

lset <- ListenerSet$new(sender="Other",dburi="mongodb://localhost",
               colname="messages",dbname="test",listeners=listeners)

mess1 <- P4Message(app="default",uid="Phred",context="Down Hill",
                  sender="EIEvent",mess="New Observables",
                  details=list(trophy="gold",solvedtime=10))

resetListeners(lset,"ALL","default")
receiveMessage(lset,mess1)
## Check recieved messages.
stopifnot(lset$messdb()$count(buildJQuery(app="default"))==1L,
          length(listeners$cl$messages)==1L,
          listeners$upd$messdb()$count(buildJQuery(app="default"))==1L,
          listeners$ups$messdb()$count(buildJQuery(app="default"))==1L,
          listeners$il$messdb()$count(buildJQuery(app="default"))==1L,
          nrow(listeners$tl$returnDF())==1L)

resetListeners(lset,c("Self","cl","il","tl"),"default")
stopifnot(lset$messdb()$count(buildJQuery(app="default"))==0L,
          length(listeners$cl$messages)==0L,
          listeners$upd$messdb()$count(buildJQuery(app="default"))==1L,
          listeners$ups$messdb()$count(buildJQuery(app="default"))==1L,
          listeners$il$messdb()$count(buildJQuery(app="default"))==0L,
          nrow(listeners$tl$returnDF())==0L)

resetListeners(lset,"ALL","default")
stopifnot(lset$messdb()$count(buildJQuery(app="default"))==0L,
          length(listeners$cl$messages)==0L,
          listeners$upd$messdb()$count(buildJQuery(app="default"))==0L,
          listeners$ups$messdb()$count(buildJQuery(app="default"))==0L,
          listeners$il$messdb()$count(buildJQuery(app="default"))==0L,
          nrow(listeners$tl$returnDF())==0L)



## End(Not run)
```

`resetProcessedMessages` *Clears the processed flags on the matching messages*

Description

The MessageQueue class uses the processed field of the P4Message object to indicate which messages have been processed. This method clears the processed flag, so that messages can be reprocessed.

Usage

```
resetProcessedMessages(queue, repquery)
## S4 method for signature 'MongoQueue'
resetProcessedMessages(queue, repquery)
## S4 method for signature 'ListQueue'
resetProcessedMessages(queue, repquery)
```

Arguments

queue	An object of class MessageQueue
repquery	A list giving a mongo query (see buildJQuery). Only messages matching the query will be reprocessed. [Currently, the ListQueue method ignores this argument.]

Details

When operating on a MongoQueue, an update query is run which sets the processed field of the messages to FALSE. The repquery is used to unmark a subset of messages.

For the ListQueue method, all messages are unmarked regardless of the query.

Value

Function run for side effects, result is status information.

Note

The current ListQueue implementation is pretty minimal, and will probably get updated.

Author(s)

Russell Almond

See Also

MessageQueue, markAsProcessed, fetchNextMessage

Examples

```
## Writeme
```

serializeData *Produces a string with a JSON representation of an R object*

Description

This function wraps the serializeJSON with encodeString to properly quote internal quotes. This *slob* (string large object) can be stored in a database. In particular, it is the default method for the jsonEncoder field of the UpdateListener object.

Usage

```
serializeData(jlist)
```

Arguments

jlist	A list containing the data to be serialized.

Value

A quoted string containing the JSON representation of the argument.

Author(s)

Russell Almond

See Also

serializeJSON, encodeString, UpdateListener

Examples

```
dat <- list(response="b", score=1)
serializeData(dat)
```

TableListener-class *Class "TableListener"*

Description

A listener that captures data from a P4Message and puts it into a dataframe.

Details

This listener builds up a data frame with selected data from the messages. What data is captured is controlled by the `fieldlist` object. This is a named character vector whose names correspond to field names and whose values correspond to type names (see `typeof`. The type can also be one of the two special types, `ordered` or `factor`. The following is a summary of the most common types:

`"numeric"`, `"logical"`, `"integer"`, `"double"`: These are numeric values.

`"character"`: These are character values. They are not converted to factors (see factor types below).

`"list"`,`"raw"`, **other values returned by** `typeof`: These are usuable, but should be used with caution because the output data frame may not be easy to export to other program.

`"ordered(...)"`, `"factor(...)"`: These produce objects of type `ordered` and `factor` with the comma separated values between the parenthesis passed as the `levels` argument. For example, `"ordered(Low,Medium,High)"` will produces an ordered factor with three levels. (Note that levels should be in increasing order for ordered factors, but this doesn't matter for unordered factors.)

For most fields, the field name is matched to the corresponding element of the `details` of the messages. The exceptions are the names `app`, `context`, `uid`, `mess`, `sender`, `timestamp`, which return the value of the corresponding header fields of the message. Note that

Extends

This class implements the `Listener` interface.

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

isListener `signature(x = "TableListener")`: TRUE

receiveMessage `signature(x = "TableListener")`: If the message is in the `messSet`, it adds a row to its internal table using the fields specified in `fieldlist`. (See details.)

listenerName `signature(x = "TableListener")`: returns the name of the table. This is usually also the filename where the table will be stored.

listenerDataTable `signature(listener = "TableListener", appid )`: Builds a data datable from the messages.

Data Table

When the `listenerDataTable` method is called, the table just returns the internal table.

Fields

`name`: Object of class `character` naming the listener.

`fieldlist`: A named `character` vector giving the names and types of the columns of the output matrix. See details.

`df`: Object of class `data.frame` this is the output data frame. Note that the first line is blank line. Use the function `$returnDF()` to get the valid rows.

`messSet`: A vector of class `character` giving the name of messages which are sent to the database. Only messages for which `mess(mess)` is an element of `messSet` will be added to the table..

Class-Based Methods

`receiveMessage(mess)`: Processes the message argument.

`initDF()`: An internal function that sets up the first row of the table as a blank line of the proper types. Called by `receiveMessage()`.

`initialize(name, fieldlist, messSet, ...)`: Initializes the fields.

`reset(app)`: Calls `initDF()` to reset the table.

`returnDF()`: Returns the part of the `df` which has data (e.g., omits first line which is used to set the types.)

Author(s)

Russell Almond, Lukas Liu, Nan Wang

References

This is an example of the observer design pattern. https://en.wikipedia.org/wiki/Observer_pattern.

See Also

Listener, P4Message, UpdateListener, InjectionListener, CaptureListener, UpsertListener, TableListener,

Examples

```
mess1 <- P4Message(app="default",uid="Phred",context="Down Hill",
                   sender="EIEvent",mess="New Observables",
                   as.POSIXct("2018-11-04 21:15:25 EST"),
                   details=list(trophy="gold",solvedtime=10))
mess2 <- P4Message(app="default",uid="Phred",context="Around the Tree",
                   sender="EIEvent",mess="New Observables",
                   as.POSIXct("2018-11-04 21:16:35 EST"),
                   details=list(trophy="silver",solvedtime=25))
tabMaker <- TableListener(name="Trophy Table",
                   messSet="New Observables",
                   fieldlist=c(uid="character", context="character",
                               timestamp="character",
                               solvedtime="numeric",
                               trophy="ordered(none,silver,gold)"))

receiveMessage(tabMaker,mess1)
tabMaker$returnDF()
```

UpdateListener-class *Class* "UpdateListener"

Description

This Listener updates an existing record (in a Mongo collection) for the student (uid), with the contents of the data (details) field of the message.

Details

The database is a mongo collection identified by dburi, dbname and colname (collection within the database). The mess field of the P4Message is checked against the applicable messages in messSet. If it is there, then the record in the database corresponding to the qfields (by default app(mess) and uid(mess)) is updated. Specifically, the field targetField is set to details(mess). The function jsonEncoder is called to encode the target field as a JSON object for injection into the database.

If targetField="", then the behavior is slightly different. Instead the fields in details(mess) (labeled by their names) are updated.

Extends

This class implements the Listener interface.

All reference classes extend and inherit methods from "envRefClass".

Methods

isListener signature(x = "UpdateListener"): TRUE

receiveMessage signature(x = "UpdateListener", message): If the message is in the messSet, it updates the record corresponding to app(mess) and uid(mess) in the database with the contents of details(mess). (See details.)

listenerName signature(x = "UpdateListener"): Returns the name assigned to the listener.

listenerDataTable signature(listener = "UpdateListener", appid): Builds a data datable from the messages.

Data Table

When the listenerDataTable method is called, the table is made by applying mdbFind to the target column. The behavior is different depending on whether or not a targetField is specified. If there is no target field, then all fields of the column are returned.

If there is a targetField, then the jsonDecoder function is applied to its value, and it is joined with the app, uid, context, timestamp fields from the header to make the data table.

Fields

`dbname`: Object of class `character` giving the name of the Mongo database

`dburi`: Object of class `character` giving the url of the Mongo database.

`colname`: Object of class `character` giving the column of the Mongo database.

`messSet`: A vector of class `character` giving the name of messages which are sent to the database. Only messages for which `mess(mess)` is an element of `messSet` will be inserted.

`db`: Object of class `MongoDB` giving the database. Use `messdb()` to access this field to makes sure it has been set up.

`qfields`: Object of class `character` giving the names of the fields which should be considered a key for the messages.

`targetField`: Object of class `character` naming the field which is to be set.

`jsonEncoder`: A `function` or a non-empty `character` scalar naming a function which will be used to encode `details(mess)` as a JSON object. The default is `unparseData`.

`decoderEncoder`: A `function` or `character` scalar naming a function which will be used to decode the target field when building a data table. The default is `parseData`.

Class-Based Methods

`messdb()`: Accessor for the database collection. Initializes the connection if it has not been set up.

`receiveMessage(mess)`: Does the work of updating the database. See Details.

`reset(app)`: Empties the database collection of messages with this app id.

`initialize(sender, dbname, dburi, colname, messSet, ...)`: Sets default values for fields.

Author(s)

Russell Almond

References

This is an example of the observer design pattern. https://en.wikipedia.org/wiki/Observer_pattern.

See Also

`Listener`, `P4Message`, `UpdateListener`, `InjectionListener`, `CaptureListener`, `UpsertListener`, `TableListener`, `mongo`

The function `unparseData` is the default encoder.

Examples

```
## Updating the data field.
fm <- mongo::fake_mongo(count=list(0L,1L))
ul <- UpdateListener("tester", db=fm,messSet=c("Scored Response"))

## New message, insert
  mess1 <- P4Message("Fred","Task 1","Evidence ID","Scored Response",
```

```
                    as.POSIXct("2018-11-04 21:15:25 EST"),
                    list(correct=TRUE,selection="D"))
receiveMessage(ul,mess1)

## Message is update, update.
mess1a <- P4Message("Fred","Task 1","Evidence ID","Scored Response",
                    as.POSIXct("2018-11-04 21:15:40 EST"),
                    list(correct=TRUE,selection="D",key="D"))
receiveMessage(ul,mess1a)

### No target field, details are added to record.

data2json <- function(dat) {
  jsonlite::toJSON(mongo::unboxer(dat))
}

upwind <- UpdateListener(messSet=c("Money Earned","Money Spent"),
                         db=mongo::MongoDB("Players",noMongo=TRUE),
                         targetField="",
                         jsonEncoder="data2json")

mess2 <- P4Message(app="default",uid="Phred",context="Down Hill",
                   sender="EIEvent",mess="Money Earned",
                   details=list(trophyHall=list(list("Down Hill"="gold"),
                                                list("Stairs"="silver")),
                                bankBalance=10))

receiveMessage(upwind,mess2)
```

`UpsertListener-class` *Class* "`UpsertListener`"

Description

This listener takes messages that match its incomming set and inject them into another Mongo database (presumably a queue for another service). If a matching message exists, it is replaced instead.

Details

The database is a `mongo` collection identified by `dburi`, `dbname` and `colname` (collection within the database). The `mess` field of the `P4Message` is checked against the applicable messages in `messSet`. If it is there, then the message is saved in the collection.

Before the message is saved, the collection is checked to see if another message exits which matches on the fields listed in `qfields`. If this is true, the message in the database is replaced. If not, the message is inserted.

Extends

This class implements the `Listener` interface.

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

isListener `signature(x = "UpsertListener")`: returns true.

receiveMessage `signature(x = "UpsertListener", message)`: If the message is in the `messSet`, it saves or replaces the message inthe database. (See details)

listenerName `signature(x = "UpsertListener")`: Returns the name assigned to the listener.

listenerDataTable `signature(listener = "UpsertListener", appid )`: Builds a data datable from the messages.

Data Table

When the `listenerDataTable` method is called, a general find query (`mdbFind` on the backing collection. The `app`, `uid`, `context`, `timestamp` fields are selected, and the `data` (details) field is unpackaged and added as additional columns.

Fields

`sender`: Object of class `character` which is used as the sender field for the message.

`dbname`: Object of class `character` giving the name of the Mongo database

`dburi`: Object of class `character` giving the url of the Mongo database.

`colname`: Object of class `character` giving the column of the Mongo database.

`qfields`: Object of class `character` giving the names of the fields which should be considered a key for the messages.

`messSet`: A vector of class `character` giving the name of messages which are sent to the database. Only messages for which `mess(mess)` is an element of `messSet` will be inserted.

`db`: Object of class `MongoDB` giving the database. Use `messdb()` to access this field to makes sure it has been set up.

Class-Based Methods

`messdb()`: Accessor for the database collection. Initializes the connection if it has not been set up.

`receiveMessage(mess)`: Does the work of inserting the message. See Details.

`reset(app)`: Empties the database collection of messages with this app id.

`initialize(sender, dbname, dburi, colname, messSet, qfields, ...)`: Sets the default values for the fields.

Author(s)

Russell Almond

References

This is an example of the observer design pattern. https://en.wikipedia.org/wiki/Observer_pattern.

See Also

Listener, P4Message, UpsertListener, UpdateListener, CaptureListener, InjectionListener, TableListener, mongo

Examples

```
## Not run:
mess1 <- P4Message(app="default",uid="Phred",context="Down Hill",
                   sender="EABN",mess="Statistics",
                   details=list("Physics_EAP"=0.5237,"Physics_Mode"="High"))
ul <- UpsertListener(colname="Statistics",qfields=c("app","uid"),
         messSet=c("Statistics"))
receiveMessage(ul,mess1)

## End(Not run)
```

`withFlogging` *Invoke expression with errors logged and traced*

Description

This is a version of `try` with a couple of important differences. First, error messages are redirected to the log, using the `flog.logger` mechanisms. Second, extra context information can be provided to aid with debugging. Third, stack traces are added to the logs to assist with later debugging.

Usage

```
withFlogging(expr, ..., context = deparse(substitute(expr)),
             loggername = flog.namespace(),
             tracelevel = c("WARN", "ERROR", "FATAL"))
```

Arguments

`expr`	The expression which will be exectued.
`...`	Additional context arguments. Each additional argument should have an explicit name. In the case of an error or warning, the additional context details will be added to the log.
`context`	A string identifying the context in which the error occurred. For example, it can identify the case which is being processed.
`loggername`	This is passed as the `name` argument to `flog.logger`. It defaults to the package in which the call to `withFlogging` was made.
`tracelevel`	A character vector giving the levels of conditions for which stack traces should be added to the log. Should be strings with values "TRACE", "DEBUG", "INFO", "WARN", "ERROR" or "FATAL".

Details

The various processes of the four process assessment design are meant to run as servers. So when errors occur, it is important that they get logged with sufficient detail that they can be reproduced, fixed and added to the test suite to prevent recurrance.

First, signals are caught and redirected to the appropriate `flog.logger` handler. This has several important advantages. First, the output can be directed to various files depending on the origin package. In general, the name of the package should be the name of the logger. So, `flog.appender(appender.file("/var/log/Proc4/EIEvent_log.json"), name="EIEvent")` would log error from the EIEvent package to the named file. Furthermore, `flog.layout(layout.json,name="EIEvent")` will cause the log to be in JSON format.

Second, additional context information is logged at the "DEBUG" level when an condition is signaled. The `context` string is printed along with the error or warning message. This can be used, for example, to provide information about the user and task that was being processed when the condition was signaled. In addition, any of the `...` arguments are printed. This can be used to print information about the message being processed and the initial state of the system, so that the error condition can be reproduced.

Third, if the class of the exception is in the `tracelevel` list, then a stack trace will be logged (at the "DEBUG" level) along with the error. This should aid debugging.

Fourth, in the case of an error or fatal error, an object of class `try-error` (see `try`). Among other things, this guarentees that `withFlogging` will always return control to the next statement.

Value

If `expr` executes successfully (with no errors or fatal errors) then the value of `expr` will be returned. If an error occurs during execution, then an object of class `try-error` will be returned.

Author(s)

Russell Almond

References

The code for executing the stack trace was taken from https://stackoverflow.com/questions/1975110/printing-stack-trace-and-continuing-after-error-occurs-in-r

See Also

`try`, `flog.logger`, `flog.layout`, `flog.appender`

Examples

```
## Not run:
## Setup to log to file in json format.
flog.appender(appender.file("/var/log/Proc4/Proc4_log.json"),
    name="Proc4")
flog.layout(layout.json,name="EIEvent")

## End(Not run)
```

```
xy <- withFlogging(stop("shoes untied"),context="walking",foot="left")
stopifnot(is(xy,"try-error"))


xx <- withFlogging(log(-1))
stopifnot(is.nan(xx))

withFlogging(log(-1),tracelevel=c("ERROR","FATAL"))
```

Index