Package 'Proc4'

June 22, 2019

Version 0.4-2

Date 2019/06/20

Title Four Process Assessment Database and Dispatcher

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

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

Description Extracts observables from a sequence of events.

License Artistic-2.0

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

R topics documented:

Proc4-package   *Four Process Assessment Database and Dispatcher*

Description

Extracts observables from a sequence of events.

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.

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

The jsonlite package provides minimal support for storing S4 objects in the mongo database. In particular, toJSON provides too little support and serializeJSON wraps the object in R-specific metadata which makes the data difficult for other applications to extract. Instead, Proc4 introduces a new protocol which is suitable for saving S4 classes: a generic as.json function for converting the class to JSON, and a parse*XXX* function for reversing the process.

The as.json function calls attributes to convert the S4 object into a list, and then calls the function as.jlist to massage the elements of the list for export into JSON. The function unboxer is useful for preventing elements which should be scalars from being converted into lists. The as.json function then runs the result through toJSON to get the result.

The parse*XXX* messages reverse this process. In particular, the record is retrieved from the database and converted into a list using fromJSON. The parsing function is then called on the result to build the object. The function parseMessage provides an example. The function cleanMessageJlist does much of the interior work of the parsing and is intended for subclasses of P4Message. The getOneRec and getManyRecs functions take a parse*XXX* function as an argument to construct objects from the database.

Logging

The logging system for the Proc4 processes is mostly just the flog.logger protocol. Aside from importing the futile.logger package, Proc4 makes one addition. 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.

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 ListenerConstructors or the individual 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.

The ListenerSet class is a mixin to associate a collection of listeners with an object (the EIEngine and EAEngine 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.

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.

See Also

flog.logger, EIEvent, EABN

---

as.json   *Converts P4 messages to JSON representation*

Description

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

Usage

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

Arguments

x	An (S4) object to be serialized.
obj	The object being serialized
ml	A list of fields of the object.
serialize	A logical flag. If true, serializeJSON is used to protect the data field (and other objects which might contain complex R code.

Details

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

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

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

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

Value

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

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

Author(s)

Russell Almond

See Also

In this package: parseMessage, saveRec, parseData

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

Examples

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

## Not run:
## This is the method for P4 Messages.
setMethod("as.jlist",c("P4Message","list"), function(obj,ml) {
  ml$"_id" <- NULL
  ml$class <-NULL
  ## Use manual unboxing for finer control.
  ml$app <- unboxer(ml$app)
  ml$uid <- unboxer(ml$uid)
  if (!is.null(ml$context) && length(ml$context)==1L)
    ml$context <- unboxer(ml$context)
  if (!is.null(ml$sender) && length(ml$sender)==1L)
    ml$sender <- unboxer(ml$sender)
  if (!is.null(ml$mess) && length(ml$mess)==1L)
    ml$mess <- unboxer(ml$mess)
  ml$timestamp <- unboxer(ml$timestamp) # Auto_unboxer bug.
  ## Saves name data; need recursvie version.
  ml$data <- unparseData(ml$data)
  ml
  })

## End(Not run)
```

---

buildJQuery   *Transforms a query into JQuery JSON.*

Description

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

Usage

```
buildJQuery(..., rawfields = character())
buildJQterm(name,value)
```

Arguments

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

Details

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

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

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

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

Value

The function `buildJQuery` returns a unicode string which contains the JSON query document. The function `buildJQterm` returns a unicode string with just one field in the query document.

Author(s)

Russell Almond

References

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

See Also

as.json, parseMessage, getOneRec, getManyRecs mongo

Examples

```
## Low level test of the JQterm possibilities for fields.

stopifnot(buildJQterm("uid","Fred")=='"uid":"Fred"')
stopifnot(buildJQterm("uid",c("Phred","Fred"))=='"uid":{"$in":["Phred","Fred"]}')
time1 <- as.POSIXct("2018-08-16 19:12:19 EDT")
stopifnot(buildJQterm("time",time1)=='"time":{"$date":1534461139000}')
time1l <- as.POSIXlt("2018-08-16 19:12:19 EDT")
stopifnot(buildJQterm("time",time1l)=='"time":{"$date":1534461139000}')
time2 <- as.POSIXct("2018-08-16 19:13:19 EDT")
stopifnot(buildJQterm("time",c(time1,time2))==
          '"time":{"$in":[{"$date":1534461139000},{"$date":1534461199000}]}')
stopifnot(buildJQterm("time",c(gt=time1))==
          '"time":{ "$gt":{"$date":1534461139000} }')
stopifnot(buildJQterm("time",c(lt=time1))==
          '"time":{ "$lt":{"$date":1534461139000} }')
stopifnot(buildJQterm("time",c(gte=time1))==
          '"time":{ "$gte":{"$date":1534461139000} }')
stopifnot(buildJQterm("time",c(lte=time1))==
          '"time":{ "$lte":{"$date":1534461139000} }')
stopifnot(buildJQterm("time",c(ne=time1))==
          '"time":{ "$ne":{"$date":1534461139000} }')
stopifnot(buildJQterm("time",c(eq=time1))==
          '"time":{ "$eq":{"$date":1534461139000} }')
stopifnot(buildJQterm("time",c(gt=time1,lt=time2))==
          '"time":{ "$gt":{"$date":1534461139000}, "$lt":{"$date":1534461199000} }')
stopifnot(buildJQterm("count",c(nin=1,2:4))==
          '"count":{"$nin":[1,2,3,4]}')
stopifnot(buildJQterm("count",c("in"=1,2:4))==
          '"count":{"$in":[1,2,3,4]}')
stopifnot(buildJQterm("count",c(ne=1,ne=5))==
          '"count":{ "$ne":1, "$ne":5 }')

## Some Examples of buildJQuery on complete queries.

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


## Queries on IDs need special handling
stopifnot(buildJQuery("_id"=c(oid="123456789abcdef"))==
          '{ "_id":{ "$oid":"123456789abcdef" } }')
```

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)

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.

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, mongo

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()))
```

---

getOneRec   *Fetches Messages from a Mongo databas*

Description

This function fetches P4Message objects from a mongo database. The message parser is passed as an argument, allowing it to fetch other kinds of objects than P4Messages. The function getManyRecs retrieves all matching objects and the function getOneRec retrieves the first matching object.

Usage

```
getOneRec(jquery, col, parser, sort = c(timestamp = -1))
getManyRecs(jquery, col, parser, sort = c(timestamp = 1), limit=0)
```

Arguments

jquery   A string providing a Mongo JQuery to select the appropriate records. See
             buildJQuery.

col   A mongo collection object to be queried.

parser   A function which will take the list of fields returned from the database and build
             an appropriate R object. See parseMessage.

sort   A named numeric vector giving sorting instructions. The names should corre-
             spond to fields of the objects, and the values should be positive or negative one for
             increasing or decreasing order. Use the value NULL to leave the results unsorted.

limit   A numeric scalar giving the maximum number of objects to retrieve. If 0, then
             all objects matching the query will be retrieved.

Details

This function assumes that a number of objects (usually, but not necessarily subclasses of [P4Message](#)
objects) have been stored in a Mongo database. The col argument is the [mongo](#) object in which they
are stored. These functions retrive the selected objects.

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

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

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

Value

The function getOneRec returns an object whose type is determined by the output of the parser
function. If [parseMessage](#) is used, this will be a [P4Message](#) object.

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

Author(s)

Russell Almond

References

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

See Also

[saveRec](#), [parseMessage](#), [getOneRec](#), [getManyRecs](#) [mongo](#)

Examples

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

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))
```

```
    testcol <- mongo("Messages",
                     url="mongodb://test:secret@127.0.0.1:27017/test")
## Mongodb is the protocol
## user=test, password =secret
## Host = 127.0.0.1 -- localhost
## Port = 27017 -- Mongo default
## db = test
## collection = Messages
## collection = Messages
## Execute in Mongo Shell
## db.createUser({
## ... user: "test",
## ... pwd: "secret",
## ... roles: [{role: "readWrite", db: "test"}]
## ... });



m1 <- saveRec(m1,testcol)
m2 <- saveRec(m2,testcol)
m3 <- saveRec(m3,testcol)

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

## Note use of oid keyword to fetch object by Mongo ID.
m1a <- getOneRec(buildJQuery("_id"=c(oid=m1@"_id")),testcol,parseMessage)
stopifnot(all.equal(m1,m1a))

m123 <- getManyRecs(buildJQuery(uid="Fred"),testcol,parseMessage)
m23 <- getManyRecs(buildJQuery(uid="Fred",sender=c("EI","EA")),
                   testcol,parseMessage)
m321 <- getManyRecs(buildJQuery(uid="Fred",timestamp=c(lte=Sys.time())),
           testcol,parseMessage,sort=c(timestamp=-1))
getManyRecs(buildJQuery(uid="Fred",
                        timestamp=c(gte=Sys.time()-as.difftime(1,units="hours"))),
                        testcol,parseMessage)

## 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"): If the message is in the messSet, it saves the message to the database. (See details)

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(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.

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](#), [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, mess)
isListener(x)
## S4 method for signature 'ANY'
isListener(x)
```

Arguments

x	A object of the virtual class Listner.
mess	A [P4Message](#) which is being transmitted.

Details

The Listener class is a virtual 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 Listener is a virtual class, it does not have a formal 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 receiveMessage function is typically invoked for side effects and it may have any return value.

Author(s)

Russell Almond

References

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

See Also

Implementing Classes: CaptureListener, UpdateListener, UpsertListener, InjectionListener,

Related Classes: ListenerSet, P4Message

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)
```

ListenerConstructors   *Constructors for Listener Classes*

Description

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

Usage

```
CaptureListener(messages = list(), ...)
InjectionListener(sender = "sender", dbname = "test",
    dburi = "mongodb://localhost",
    messSet = character(), colname = "Messages", ...)
UpdateListener(dbname = "test", dburi = "mongodb://localhost",
    messSet = character(), colname = "Messages", targetField = "data",
    qfields = c("app", "uid"), jsonEncoder = "unparseData", ...)
UpsertListener(sender = "sender", dbname = "test",
    dburi = "mongodb://localhost",
    messSet = character(), colname = "Messages",
    qfields = c("app", "uid"), ...)
```

Arguments

messages	A list into which to add the messages.
sender	A character value used as the sender field of the message.
dbname	A character value giving the name of the database in which to put the message. See mongo.
dburi	A character vector giving the URI for the database. See mongo.
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.
colname	The name of the database column into which the messages will be sent. See mongo.
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.
qfields	The fields that will be used as a key when trying to find matching messages in the database for the UpsertListener.
...	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.

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, ListenerSet, mongo

Examples

```
cl <- CaptureListener()

il <- InjectionListener(sender="EI_app",
           dbname="EARecords",dburi="mongodb://localhost",
           colname="EvidenceSets",messSet="New Observables")

upsl <- UpsertListener(sender="EI_app",
           dbname="EARecords",dburi="mongodb://localhost",
           colname="LatestEvidence",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(dbname="Proc4",dburi="mongodb://localhost",
            colname="Players",targetField="data",
            messSet=c("Money Earned","Money Spent"),
            jsonEncoder="trophy2json")
```

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".

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().

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.

colname: Object of class character: the name of the column 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.

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

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,

Examples

showClass("ListenerSet")

---

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(mess, col)
markAsError(mess, col, e)
processed(x)
processingError(x)
```

Arguments

mess	An object of class [P4Message](#) to be modified.
col	A [mongo](#) collection where the message queue is stored.
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 [mongo](#) collection of messages can serve as a queue. 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.

A typical query (this example is taken from the [EIEvent-package](#)) is getOneRec(buildJQuery(app=app, processed=FALS
Here the [buildJQuery](#) call searches for unprocessed events corresponding to a particular [app](#). The sort argument ensures that the records will be sorted in ascending order according to [timestamp](#). In this example eventdb() in an internal method which returns the event collection, and [parseEvent](#) create event objects (which are a subclass of [P4Message](#).

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

Examples

```
col <- 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))
while (!is.null(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)))
```

MongoDB–class   *Class* "MongoDB"

Description

An S4-style class for the [mongo](#) class. Note that this is actually a class union, allowing for NULL if the database is not yet initialized.

Objects from the Class

NULL is an object of this class.

Objects of this class can be created with calls to [mongo](#).

Methods

No methods defined with class "MongoDB" in the signature.

Note

The original [mongo](#) class is an S3 class. Rather than just call [setOldClass](#) and exposing that, I've explosed a class union ([setClassUnion](#)) with the mongo class and NULL.

A typical usage would have this type used in the slot of an object, which would initialize the value to NULL, and then set it to a mongo object when the database connection is openned.

Author(s)

Russell Almond

See Also

[ListenerSet](#), [mongo](#)

Examples

```
showClass("MongoDB")
showClass("ListenerSet")
lset <- ListenerSet$new()
lset$messdb
```

---

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, mess)
```

Arguments

sender	An object which sends messages.
mess	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", process
m_id(x)
app(x)
uid(x)
mess(x)
context(x)
sender(x)
timestamp(x)
details(x)
## 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.

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 [jsonlite-package](#)) or saved in the Mongo database (using the [mongo](#)lite 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 [parseMessage](#), [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 parseMessage, saveRec, getOneRec

Examples

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

---

parseMessage   *Converts a JSON object into a P4 Message*

Description

The parseMessage 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. The function parseData is a helper function for parsing the data field of the P4Message object, and unparseData is its inverse.

Usage

```
parseMessage(rec)
cleanMessageJlist(rec)
parseData(messData)
parseSimpleData(messData)
unparseData(data, serialize=TRUE)
```

Arguments

rec	A named list containing JSON data.
messData	A named list containing JSON data.
data	An R object to be serialized.
serialize	A logical flag. If true, serializeJSON is used to protect the data field (and other objects which might contain complex R code.

Details

The $iterator() method of the mongo object returns a list containing the fields of the JSON object with a *name*=*value* format. This is the rec argument. The parseMessage function takes the fields of the JSON object and uses them to populate a corresponding P4Message object. Usually, some cleaning is done first (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 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 parseSimpleMessage 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 parseMessage returns a [P4Message](#) object populated with fields from the rec argument. The function cleanMessageJlist returns the cleaned rec argument.

The function unparseData returns a JSON string representing the data. The functions parseData and parseSimpleData return a list containing the data.

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](#)

[mongo](#), [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 <- parseMessage(ununboxer(as.jlist(m1,attributes(m1))))
m2a <- parseMessage(ununboxer(as.jlist(m2,attributes(m2))))
m3a <- parseMessage(ununboxer(as.jlist(m3,attributes(m3))))

ev1a <- parseMessage(ununboxer(as.jlist(ev1,attributes(ev1))))

stopifnot(all.equal(m1,m1a),
          all.equal(m2,m2a),
          all.equal(m3,m3a),
          all.equal(ev1,ev1a))
```

```
## Not run:  #Requires test DB setup.
testcol <- mongo("Messages",
                 url="mongodb://test:secret@127.0.0.1:27017/test")
## Mongodb is the protocol
## user=test, password =secret
## Host = 127.0.0.1 -- localhost
## Port = 27017 -- Mongo default
## db = test
## collection = Messages
testcol$remove('{}')  ## Clear everything for test.

m1 <- saveRec(m1,testcol)
m2 <- saveRec(m2,testcol)
m3 <- saveRec(m3,testcol)
ev1 <- saveRec(ev1,testcol)

m1 <- saveRec(m1,testcol)
m1b <- getOneRec(buildJQuery("_id"=c("oid"=m1@"_id")),testcol,parseMessage)
stopifnot(all.equal(m1,m1b))
m23 <- getManyRecs(buildJQuery("uid"="Fred",sender=c("EI","EA")),
                   testcol,parseMessage)
stopifnot(length(m23)==2L)
ev1b <- getOneRec(buildJQuery("uid"="Phred"),
                  testcol,parseMessage)
stopifnot(all.equal(ev1,ev1b))


## End(Not run)
```

---

saveRec   *Saves a P4 Message object to a Mongo database*

Description

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

Usage

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

Arguments

mess	The message (object) to be saved.
col	A mongo collection object, produced with a call to `mongo()`.
serialize	A logical flag. If true, `serializeJSON` is used to protect the `data` field (and other objects which might contain complex R code.

Value

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

Author(s)

Russell Almond

See Also

as.json, P4Message, parseMessage, getOneRec, mongo

Examples

```
## Not run: ## Need to set up database or code won't run.
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))

testcol <- mongo("Messages",
                 url="mongodb://test:secret@127.0.0.1:27017/test")
## Mongodb is the protocol
## user=test, password =secret
## Host = 127.0.0.1 -- localhost
## Port = 27017 -- Mongo default
## db = test
## collection = Messages

## Save them back to capture the ID.
m1 <- saveRec(m1,testcol)
m2 <- saveRec(m2,testcol)
m3 <- saveRec(m3,testcol)




## End(Not run)
```

---

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

Description

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

Usage

```
unboxer(x)
ununboxer(x)
```

Arguments

x   Object to be boxed/unboxed.

Details

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

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

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

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

Value

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

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

Warning: Dependence on jsonlite implementation

These functions currently rely on some internal mechanisms of the jsonline pacakge. In particular, it uses the internal function jsonlite:::as.scalar, and ununbox relies on the "scalar" class mechanism.

Note

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

Author(s)

Russell Almond

See Also

unbox, toJSON, as.jlist, parseMessage

Examples

```
## as.jlist method shows typical use of unboxer.
getMethod("as.jlist",c("P4Message","list"))

## Use ununboxer to test as.jlist/parseMessage pair.
m4 <- P4Message("Phred","Task1","PP","New Stats",
                details=list("agents"=c("ramp","ramp","lever")))
m4jl <- as.jlist(m4,attributes(m4))
m4a <- parseMessage(ununboxer(m4jl))
stopifnot(all.equal(m4,m4a))
```

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.

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"): 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.)

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: Object of class `character` naming a function which will be used to encode `details(mess)` as a JSON object. The default is unparseData.

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.

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, mongo

The function unparseData is the default encoder.

Examples

```
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))
data2json <- function(dat) {
  toJSON(sapply(dat,unboxer))
}
upwind <- UpdateListener(messSet=c("Money Earned","Money Spent"),
                         targetField="data",colname="Players",
```

```
                                jsonEncoder="data2json")

  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"): If the message is in the messSet, it saves or replaces the message inthe database. (See details)

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.

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,
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](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
```

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