Package 'Proc4'

January 25, 2019

Version 0.2-1

Date 2019/01/25

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 http://pluto.coe.fsu.edu/Proc4

R topics documented:

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 arugments 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" } }')
```

---

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 correspond to fields of the objects, and the values should be positive or negative one for increasing or decreasing order. Use the value NULL to leave the results unsorted.
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)
```

`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

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")

lset$notifyListeners(mess2)


## 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"`.

Methods

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

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

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.

messdb: Object of class mongo which is a handle to the collection where messages are logged.

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. In particular, it call mongo(colnam,uri=dburi) to open the collection for logging.

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

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, flog.logger, mongo, P4Message

Examples

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

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")
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)
```

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.
x	A message object to be queried, or converted to a string.
...	Addtional arguments for show.
object	A message object to be converted to a string.

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 `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.

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.

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,seletion="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"
)
```

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 `jsonlite-package`) or saved in the Mongo database (using the `mongolite` package).

Objects from the Class

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

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.

Methods

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.

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)
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.

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 `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))
```

---

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
```

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 printed 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 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