Package 'MazamaWebUtils'

September 29, 2018

Type Package

Version 0.1.7

Title Utility Functions for Building Web Databrowsers

Author Jonathan Callahan [aut, cre]

Maintainer Jonathan Callahan <jonathan.s.callahan@gmail.com>

Depends R (>= 3.1.0),

Imports dplyr, futile.logger, lubridate, mime, stringr, webutils

Suggests knitr, markdown, testthat

Description A suite of utility functions providing standardized functionality often needed in web services including: logging, cache management and parsing of http request headers.

License GPL-3

URL https://github.com/MazamaScience/MazamaWebUtils

BugReports https://github.com/MazamaScience/MazamaWebUtils/issues

Encoding UTF-8

LazyData true

RoxygenNote 6.1.0

NeedsCompilation no

Repository CRAN

Date/Publication 2018-09-29 19:50:03 UTC

R topics documented:

`cgiRequest` *Create a CGI Request Object*

Description

A request object is created from the appropriate environment variables and is returned as a list. List elements include:

- params – list of request parameters
- headers – list of HTTP headers
- method – "GET"
- raw – NULL
- content_type – NULL
- protocol – "http"
- body – NULL

Usage

```
cgiRequest(testParams = NULL)
```

Arguments

`testParams`	URL request parameters for testing GET requests

Details

Even in the modern era (>= 2017) it is still sometimes useful to build simple web services using CGI scripts. Benefits include: ease of coding; use of standard port 80; service uptime: even if the CGI script dies while handling an earlier request, the script will be restarted for the next request.

Using this function, the body of an R CGI script can begin with:

```
req <- cgiRequest()
headers <- req$params
params <- req$params
...
```

Value

A list containing CGI request elements

Note

The returned object mimics the request object created in the **jug** package.

References

https://github.com/Bart6114/jug/blob/master/R/request.R

Examples

```
# Example borrowed from webutils::parse_query
q <- "foo=1%2B1%3D2&bar=yin%26yang"
req <- cgiRequest(q)
str(req$params)
```

`httpResponse.contentType`

Create a Content Type String

Description

The `type` parameter is typically the file extension.

Usage

```
httpResponse.contentType(type = "text")
```

Arguments

`type`	file type or standard extension

Value

A character string with the appropriate MIME type.

Examples

```
httpResponse.contentType('text')
httpResponse.contentType('json')
httpResponse.contentType('png')
```

httpResponse.header *Create a HTTP Response Header*

Description

This function will generate a header string containing only a minimal set of possible response header elements including:

- Content-Type

Usage

```
httpResponse.header(type = "text")
```

Arguments

type	file type or standard extension

Value

A character string containing a valid HTTP Response header.

Examples

```
httpResponse.header('text')
httpResponse.header('json')
httpResponse.header('png')
```

initializeLogging *Initialize Standard Log Files*

Description

Convenience function that wraps logging initialization steps common to Mazama Science web services:

```
result <- try({
  Copy and old log files
  timestamp <- strftime(lubridate::now(), "
  for ( logLevel in c("TRACE","DEBUG","INFO","ERROR") ) {
    oldFile <- file.path(logDir,paste0(logLevel,".log"))
    newFile <- file.path(logDir,paste0(logLevel,".log.",timestamp))
    if ( file.exists(oldFile) ) {
      file.rename(oldFile, newFile)
    }
  }
```

```
}, silent=TRUE)
stopOnError(result, "Could not rename old log files.")

result <- try({
  # Set up logging
  logger.setup(traceLog = file.path(logDir, "TRACE.log"),
               debugLog=file.path(logDir, "DEBUG.log"),
               infoLog=file.path(logDir, "INFO.log"),
               errorLog=file.path(logDir, "ERROR.log"))
}, silent=TRUE)
stopOnError(result, "Could not create log files.")
```

Usage

```
initializeLogging(logDir = NULL)
```

Arguments

logDir	directory in which to write log files

Value

No return value.

---

logger.debug *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `DEBUG` level log statements.

Usage

```
logger.debug(msg, ...)
```

Arguments

msg	message with format strings applied to additional arguments
...	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.error *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `ERROR` level log statements.

Usage

```
logger.error(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

`logger.fatal` *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `FATAL` level log statements.

Usage

```
logger.fatal(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

`logger.info` *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `INFO` level log statements.

Usage

```
logger.info(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

`logger.setLevel` *Set Console Log Level*

Description

By default, the looger threshold is set to `FATAL` so that the console will typically receive no log messages. By setting the level to one of the other log levels: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` users can see logging messages while running commands at the command line.

Usage

```
logger.setLevel(level)
```

Arguments

`level`	threshold level

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Set up console logging only
logger.setup()
logger.setLevel(DEBUG)

## End(Not run)
```

`logger.setup` *Set Up Python-Style Logging*

Description

Good logging allows package developers and users to create log files at different levels to track and debug lengthy or complex calculations. "Python-style" logging is intended to suggest that users should set up multiple log files for different log severities so that the `errorLog` will contain only log messages at or above the `ERROR` level while a `debugLog` will contain log messages at the `DEBUG` level as well as all higher levels.

Python-style log files are set up with `logger.setup()`. Logs can be set up for any combination of log levels. Accepting the default `NULL` setting for any log file simply means that log file will not be created.

Python-style logging requires the use of `logger.debug()` style logging statements as seen in the example below.

Usage

```
logger.setup(traceLog = NULL, debugLog = NULL, infoLog = NULL,
  warnLog = NULL, errorLog = NULL, fatalLog = NULL)
```

Arguments

`traceLog`	file name or full path where `logger.trace()` messages will be sent
`debugLog`	file name or full path where `logger.debug()` messages will be sent
`infoLog`	file name or full path where `logger.info()` messages will be sent
`warnLog`	file name or full path where `logger.warn()` messages will be sent
`errorLog`	file name or full path where `logger.error()` messages will be sent
`fatalLog`	file name or full path where `logger.fatal()` messages will be sent

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.trace logger.debug logger.info logger.warn logger.error logger.fatal

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow lot statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logger.trace *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate TRACE level log statements.

Usage

```
logger.trace(msg, ...)
```

Arguments

msg	message with format strings applied to additional arguments
...	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

logger.setup

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

`logger.warn` *Python-Style Logging Statements*

Description

After initializing the level-specific log files with `logger.setup(...)`, this function will generate `WARN` level log statements.

Usage

```
logger.warn(msg, ...)
```

Arguments

`msg`	message with format strings applied to additional arguments
`...`	additional arguments to be formatted

Value

No return value.

Note

All functionality is built on top of the excellent **futile.logger** package.

See Also

`logger.setup`

Examples

```
## Not run:
# Only save three log files
logger.setup(debugLog='debug.log', infoLog='info.log', errorLog='error.log')

# But allow log statements at all levels within the code
logger.trace('trace statement #%d', 1)
logger.debug('debug statement')
logger.info('info statement %s %s', "with", "arguments")
logger.warn('warn statement %s', "about to try something dumb")
result <- try(1/"a", silent=TRUE)
logger.error('error message: %s', geterrmessage())
logger.fatal('fatal statement %s', "THE END")

## End(Not run)
```

---

logLevels *Log Levels*

Description

Log levels matching those found in **futile.logger**. Available levels include:

`FATAL ERROR WARN INFO DEBUG TRACE`

Usage

```
FATAL
```

Format

An object of class `integer` of length 1.

---

manageCache *Manage the Size of a Cache*

Description

If `cacheDir` takes up more than `maxCacheSize` megabytes on disk, files wll be removed in order of oldest access time. Only files matching `extensions` are eligible for removal.

Usage

```
manageCache(cacheDir, extensions = c("html", "json", "pdf", "png"),
  maxCacheSize = 100)
```

Arguments

cacheDir	location of cache directory
extensions	vector of file extensions eligible for removal
maxCacheSize	maximum cache size in megabytes

Value

Invisibly returns the number of files removed.

Examples

```
# Create a cache directory and fill it with 1.6 MB of data
CACHE_DIR <- tempdir()
write.csv(matrix(1,400,500), file=file.path(CACHE_DIR,'m1.csv'))
write.csv(matrix(2,400,500), file=file.path(CACHE_DIR,'m2.csv'))
write.csv(matrix(3,400,500), file=file.path(CACHE_DIR,'m3.csv'))
write.csv(matrix(4,400,500), file=file.path(CACHE_DIR,'m4.csv'))
for (file in list.files(CACHE_DIR, full.names=TRUE)) {
  print(file.info(file)[,c(1,6)])
}
# Remove files based on last access time until we get under 1 MB
manageCache(CACHE_DIR, extensions='csv', maxCacheSize=1)
for (file in list.files(CACHE_DIR, full.names=TRUE)) {
  print(file.info(file)[,c(1,6)])
}
```

---

stopOnError *Error Message Translator*

Description

When writing R code to be used in production systems that work with user supplied input, it is important to enclose chunks of code inside of a `try()` block. It is equally important to generate error log messages that can be found and understood during an autopsy when something fails

At Mazama Science we have our own internal standard for how to do error handling in a manner that allows os to quickly navigate to the source of errors in a production system.

The example section contains a snippet of how we use this function.

Usage

```
stopOnError(result, err_msg = "")
```

Arguments

result	return from a `try()` block
err_msg	custom error message

Value

Issues a `stop()` with an appropriate error message.

Examples

```
## Not run:
logger.setup()

# Arbitrarily deep in the stack we might have:
myFunc <- function(x) {
  a <- log(x)
}

userInput <- 10
result <- try({
  myFunc(x=userInput)
}, silent=TRUE)
stopOnError(result)

userInput <- "ten"
result <- try({
  myFunc(x=userInput)
}, silent=TRUE)
stopOnError(result)

result <- try({
  myFunc(x=userInput)
}, silent=TRUE)
stopOnError(result, "Unable to process user input")

## End(Not run)
```

Index