Package: logging (via r-universe)

October 20, 2026

Version 0.10-108

Title R Logging Package

Description Pure R implementation of the ubiquitous log4j package. It offers hierarchic loggers, multiple handlers per logger, level based filtering, space handling in messages and custom formatting.

URL https://github.com/WLOGSolutions/r-logging

BugReports https://github.com/WLOGSolutions/r-logging/issues

License GPL-3

Encoding UTF-8

Language en-US

Depends R (>= 3.2.0)

Imports methods

Suggests testthat, crayon

RoxygenNote 6.1.1

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

Date/Publication 2019-08-11 15:41:01 UTC

RemoteUrl https://github.com/wlogsolutions/r-logging

RemoteRef HEAD

RemoteSha f8ca2471cdb6d8c762439d9cd5a671891e621ada

RemoteSubdir pkg

Contents

logging-package 2
bootstrapping 3
getHandler 4
getLogger 4
handlers-management 5

inbuilt-actions 6
logging-entrypoints 7
loglevels 8
resetMsgComposer 8
setLevel 9
setMsgComposer 9
updateOptions 10

Index **11**

`logging-package` *A tentative logging package.*

Description

A logging package emulating the Python logging package.

What you find here behaves similarly to what you also find in Python. This includes hierarchic loggers, multiple handlers at each logger, the possibility to specify a formatter for each handler (one default formatter is given), same levels (names and numeric values) as Python's logging package, a simple logging.BasicConfig function to quickly put yourself in a usable situation...

This package owes a lot to my employer, r-forge, the stackoverflow community, Brian Lee Yung Rowe's futile package (v1.1) and the documentation of the Python logging package.

Details

Index:

`basicConfig` bootstrapping the logging package

`addHandler` add a handler to a logger

`getLogger` set defaults and get current values for a logger

`removeHandler` remove a handler from a logger

`setLevel` set logging.level for a logger

To use this package, include logging instructions in your code, possibly like this:

```
library(logging)
basicConfig()
addHandler(writeToFile, logger="company", file="sample.log")
loginfo("hello world", logger="")
logwarn("hello company", logger="company.module")
```

The `basicConfig` function adds a console handler to the root logger, while the explicitly called `addHandler` adds a file handler to the 'company' logger. the effect of the above example is two lines on the console and just one line in the `sample.log` file.

The web pages for this package on r-forge are kept decently up to date and contain a usable tutorial. Check the references.

References

the python logging module: http://docs.python.org/library/logging.html
this package at github: https://github.com/WLOGSolutions/r-logging/

`bootstrapping` *Bootstrapping the logging package.*

Description

`basicConfig` and `logReset` provide a way to put the logging package in a know initial state.

Usage

```
basicConfig(level = 20)

logReset()
```

Arguments

`level`	The logging level of the root logger. Defaults to INFO. Please do notice that this has no effect on the handling level of the handler that basicConfig attaches to the root logger.

Details

`basicConfig` creates the root logger, attaches a console handler(by *basic.stdout* name) to it and sets the level of the handler to `level`. You must not call `basicConfig` to for logger to work any more: then root logger is created it gets initialized by default the same way as `basicConfig` does. If you need clear logger to fill with you own handlers use `logReset` to remove all default handlers.

`logReset` reinitializes the whole logging system as if the package had just been loaded except it also removes all default handlers. Typically, you would want to call `basicConfig` immediately after a call to `logReset`.

Examples

```
basicConfig()
logdebug("not shown, basic is INFO")
logwarn("shown and timestamped")
logReset()
logwarn("not shown, as no handlers are present after a reset")
```

getHandler *Retrieves a handler from a logger.*

Description

Handlers are not uniquely identified by their name. Only within the logger to which they are attached is their name unique. This function is here to allow you grab a handler from a logger so you can examine and alter it.

Typical use of this function is in `setLevel(newLevel, getHandler(...))`.

Usage

```
getHandler(handler, logger = "")
```

Arguments

handler	The name of the handler, or its action.
logger	Optional: the name of the logger. Defaults to the root logger.

Value

The retrieved handler object. It returns NULL if handler is not registerd.

Examples

```
logReset()
addHandler(writeToConsole)
getHandler("basic.stdout")
```

getLogger *Set defaults and get the named logger.*

Description

Make sure a logger with a specific name exists and return it as a *Logger* S4 object. if not yet present, the logger will be created and given the values specified in the ... arguments.

Usage

```
getLogger(name = "", ...)
```

Arguments

name	The name of the logger
...	Any properties you may want to set in the newly created logger. These have no effect if the logger is already present.

Value

The logger retrieved or registered.

Examples

```
getLogger()  # returns the root logger
getLogger('test.sub')  # constructs a new logger and returns it
getLogger('test.sub')  # returns it again
```

---

handlers-management *Add a handler to or remove one from a logger.*

Description

Use this function to maintain the list of handlers attached to a logger.

`addHandler` and `removeHandler` are also offered as methods of the *Logger* S4 class.

Usage

```
addHandler(handler, ..., logger = "")

removeHandler(handler, logger = "")
```

Arguments

`handler`	The name of the handler, or its action
`...`	Extra parameters, to be stored in the handler list ... may contain extra parameters that will be passed to the handler action. Some elements in the ... will be interpreted here.
`logger`	the name of the logger to which to attach the new handler, defaults to the root logger.

Details

Handlers are implemented as environments. Within a logger a handler is identified by its *name* and all handlers define at least the three variables:

level all records at level lower than this are skipped.

formatter a function getting a record and returning a string

`action(msg, handler)` a function accepting two parameters: a formatted log record and the handler itself. making the handler a parameter of the action allows us to have reusable action functions.

Being an environment, a handler may define as many variables as you think you need. keep in mind the handler is passed to the action function, which can check for existence and can use all variables that the handler defines.

Examples

```
logReset()
addHandler(writeToConsole)
names(getLogger()[["handlers"]])
loginfo("test")
removeHandler("writeToConsole")
logwarn("test")
```

`inbuilt-actions` *Predefined(sample) handler actions*

Description

When you define a handler, you specify its name and the associated action. A few predefined actions described below are provided.

Usage

```
writeToConsole(msg, handler, ...)

writeToFile(msg, handler, ...)
```

Arguments

`msg`	A formated message to handle.
`handler`	The handler environment containing its options. You can register the same action to handlers with different properties.
`...`	parameters provided by logger system to interact with the action.

Details

A handler action is a function that accepts a formated message and handler configuration.

Messages passed are filtered already regarding loglevel.

. . . parameters are used by logging system to interact with the action. . . . can contain *dry* key to inform action that it meant to initialize itself. In the case action should return TRUE if initialization succeded.

If it's not a dry run . . . contain the whole preformated *logging.record*. A *logging.record* is a named list and has following structure:

msg contains the real formatted message

level message level as numeric

levelname message level name

logger name of the logger that generated it

timestamp formatted message timestamp

`writeToConsole` detects if crayon package is available and uses it to color messages. The coloring can be switched off by means of configuring the handler with *color_output* option set to FALSE.

`writeToFile` action expects file path to write to under *file* key in handler options.

Examples

```
## define your own function and register it with a handler.
## author is planning a sentry client function.  please send
## any interesting function you may have written!
```

---

logging-entrypoints *Entry points for logging actions*

Description

Generate a log record and pass it to the logging system.

Usage

```
logdebug(msg, ..., logger = "")

logfinest(msg, ..., logger = "")

logfiner(msg, ..., logger = "")

logfine(msg, ..., logger = "")

loginfo(msg, ..., logger = "")

logwarn(msg, ..., logger = "")

logerror(msg, ..., logger = "")

levellog(level, msg, ..., logger = "")
```

Arguments

`msg`	the textual message to be output, or the format for the ... arguments
`...`	if present, msg is interpreted as a format and the ... values are passed to it to form the actual message.
`logger`	the name of the logger to which we pass the record
`level`	The logging level

Details

A log record gets timestamped and will be independently formatted by each of the handlers handling it.

Leading and trailing whitespace is stripped from the final message.

Examples

```
logReset()
addHandler(writeToConsole)
loginfo('this goes to console')
logdebug('this stays silent')
```

---

loglevels *The logging levels, names and values*

Description

This list associates names to values and vice versa.
Names and values are the same as in the python standard logging module.

Usage

```
loglevels
```

Format

An object of class `numeric` of length 11.

---

resetMsgComposer *Resets previously set message composer.*

Description

Resets previously set message composer.

Usage

```
resetMsgComposer(container = "")
```

Arguments

`container`	name of logger to reser message composer for (type: character)

setLevel *Set logging.level for the object.*

Description

Alter an existing logger or handler, setting its *logging.level* to a new value. You can access loggers by name, while you must use `getHandler` to get a handler.

Usage

```
setLevel(level, container = "")
```

Arguments

`level`	The new level for this object. Can be numeric or character.
`container`	a logger, its name or a handler. Default is root logger.

Examples

```
basicConfig()
setLevel("FINEST")
setLevel("DEBUG", getHandler("basic.stdout"))
```

setMsgComposer *Sets message composer for logger.*

Description

Message composer is used to compose log message out of formating string and arguments. It is function with signature `function(msg, ...)`. Formating message is passed under msg and formating arguments are passed as `...`.

Usage

```
setMsgComposer(composer_f, container = "")
```

Arguments

`composer_f`	message composer function (type: function(msg, ...))
`container`	name of logger to reser message composer for (type: character)

Details

If message composer is not set default is in use (realized with `sprintf`). If message composer is not set for sub-logger, parent's message composer will be used.

Examples

```
setMsgComposer(function(msg, ...) paste0("s-", msg, "-e"))
loginfo("a message") # will log '<TS> INFO::s-a message-e'
resetMsgComposer()
loginfo("a message") # will log '<TS> INFO::a message'
```

`updateOptions` *Changes settings of logger or handler.*

Description

Changes settings of logger or handler.

Usage

```
updateOptions(container, ...)

## S3 method for class 'character'
updateOptions(container, ...)

## S3 method for class 'environment'
updateOptions(container, ...)

## S3 method for class 'Logger'
updateOptions(container, ...)
```

Arguments

`container`	a logger, its name or a handler.
`...`	options to set for the container.

Methods (by class)

- `character`: Update options for logger identified by name.
- `environment`: Update options of logger or handler passed by reference.
- `Logger`: Update options of logger or handler passed by reference.

Index

∗ **datasets**
loglevels, 8
∗ **package**
logging-package, 2

addHandler *(*handlers-management*)*, 5

basicConfig *(*bootstrapping*)*, 3
bootstrapping, 3

getHandler, 4
getLogger, 4

handlers-management, 5

inbuilt-actions, 6

levellog *(*logging-entrypoints*)*, 7
logdebug *(*logging-entrypoints*)*, 7
logerror *(*logging-entrypoints*)*, 7
logfine *(*logging-entrypoints*)*, 7
logfiner *(*logging-entrypoints*)*, 7
logfinest *(*logging-entrypoints*)*, 7
logging-entrypoints, 7
logging-package, 2
loginfo *(*logging-entrypoints*)*, 7
loglevels, 8
logReset *(*bootstrapping*)*, 3
logwarn *(*logging-entrypoints*)*, 7

removeHandler *(*handlers-management*)*, 5
resetMsgComposer, 8

setLevel, 9
setMsgComposer, 9

updateOptions, 10

writeToConsole *(*inbuilt-actions*)*, 6
writeToFile *(*inbuilt-actions*)*, 6