

# Package ‘lgr’

October 13, 2022

**Type** Package

**Title** A Fully Featured Logging Framework

**Version** 0.4.4

**Maintainer** Stefan Fleck <stefan.b.fleck@gmail.com>

**Description** A flexible, feature-rich yet light-weight logging framework based on 'R6' classes. It supports hierarchical loggers, custom log levels, arbitrary data fields in log events, logging to plaintext, 'JSON', (rotating) files, memory buffers. For extra appenders that support logging to databases, email and push notifications see the the package lgr.app.

**License** MIT + file LICENSE

**URL** <https://s-fleck.github.io/lgr/>

**BugReports** <https://github.com/s-fleck/lgr/issues/>

**Depends** R (>= 3.2.0)

**Imports** R6 (>= 2.4.0)

**Suggests** cli, covr, crayon, data.table, desc, future, future.apply, glue, jsonlite, knitr, rmarkdown, rotor (>= 0.3.5), rprojroot, testthat, tibble, tools, utils, whoami, yaml

**VignetteBuilder** knitr

**Encoding** UTF-8

**RoxygenNote** 7.2.1.9000

**Collate** 'Filterable.R' 'utils-sfmisc.R' 'utils.R' 'Appender.R'  
'Filter.R' 'log\_levels.R' 'LogEvent.R' 'Layout.R' 'Logger.R'  
'basic\_config.R' 'default\_functions.R' 'event\_list.R'  
'get\_logger.R' 'lgr-package.R' 'logger\_config.R'  
'logger\_index.R' 'logger\_tree.R' 'read\_json\_lines.R'  
'simple\_logging.R' 'string\_repr.R' 'use\_logger.R'  
'utils-formatting.R' 'utils-logging.R' 'utils-rd.R'  
'utils-rotor.R'

**NeedsCompilation** no

**Author** Stefan Fleck [aut, cre] (<<https://orcid.org/0000-0003-3344-9851>>)

**Repository** CRAN

**Date/Publication** 2022-09-05 22:00:02 UTC

## R topics documented:

AppenderBuffer . . . . .	3
AppenderConsole . . . . .	5
AppenderFile . . . . .	6
AppenderFileRotating . . . . .	9
AppenderFileRotatingDate . . . . .	11
AppenderFileRotatingTime . . . . .	13
AppenderMemory . . . . .	15
AppenderTable . . . . .	17
as.data.frame.LogEvent . . . . .	18
as_LogEvent . . . . .	20
basic_config . . . . .	21
CannotInitializeAbstractClassError . . . . .	22
colorize_levels . . . . .	23
default_exception_handler . . . . .	24
EventFilter . . . . .	24
event_list . . . . .	26
Filterable . . . . .	27
FilterForceLevel . . . . .	29
FilterInject . . . . .	30
get_caller . . . . .	31
get_logger . . . . .	32
get_log_levels . . . . .	33
is_filter . . . . .	34
label_levels . . . . .	34
Layout . . . . .	35
LayoutFormat . . . . .	36
LayoutGlue . . . . .	39
LayoutJson . . . . .	41
LogEvent . . . . .	43
Logger . . . . .	44
LoggerGlue . . . . .	52
logger_config . . . . .	54
logger_index . . . . .	55
logger_tree . . . . .	55
pad_right . . . . .	56
print.Appender . . . . .	57
print.LogEvent . . . . .	57
print.Logger . . . . .	59
print.logger_tree . . . . .	60
read_json_lines . . . . .	61
simple_logging . . . . .	62

standardize_threshold . . . . .	64
string_repr . . . . .	65
suspend_logging . . . . .	66
toString.LogEvent . . . . .	67
use_logger . . . . .	68
with_log_level . . . . .	68

<b>Index</b>	<b>70</b>
--------------	-----------

---

AppenderBuffer	<i>Log to a memory buffer</i>
----------------	-------------------------------

---

## Description

An Appender that Buffers LogEvents in-memory and and redirects them to other Appenders once certain conditions are met.

## Fields

appenders, set\_appenders() Like for a [Logger](#). Buffered events will be passed on to these Appenders once a flush is triggered

flush\_on\_exit, set\_flush\_on\_exit(x) TRUE or FALSE: Whether the buffer should be flushed when the Appender is garbage collected (f.e when you close R)

flush\_on\_rotate, set\_flush\_on\_rotate TRUE or FALSE: Whether the buffer should be flushed when the Buffer is full (f.e when you close R). Setting this to off can have slightly negative performance impacts.

## Super classes

[lgr::Filterable](#) -> [lgr::Appender](#) -> [lgr::AppenderMemory](#) -> AppenderBuffer

## Methods

### Public methods:

- [AppenderBuffer\\$new\(\)](#)
- [AppenderBuffer\\$flush\(\)](#)
- [AppenderBuffer\\$clear\(\)](#)
- [AppenderBuffer\\$set\\_appenders\(\)](#)
- [AppenderBuffer\\$add\\_appender\(\)](#)
- [AppenderBuffer\\$remove\\_appender\(\)](#)
- [AppenderBuffer\\$format\(\)](#)

**Method** new(): The [Layout](#) for this Appender is used only to format console output of its \$show() method.

*Usage:*

```

AppenderBuffer$new(
  threshold = NA_integer_,
  layout = LayoutFormat$new(fmt = "%L [%t] %m", timestamp_fmt = "%H:%M:%S", colors
    = getOption("lgr.colors")),
  appenders = NULL,
  buffer_size = 1000,
  flush_threshold = NULL,
  flush_on_exit = TRUE,
  flush_on_rotate = TRUE,
  should_flush = NULL,
  filters = NULL
)

```

**Method** flush(): Sends the buffer's contents to all attached Appenders and then clears the Buffer

*Usage:*

```
AppenderBuffer$flush()
```

**Method** clear(): Clears the buffer, discarding all buffered Events

*Usage:*

```
AppenderBuffer$clear()
```

**Method** set\_appenders(): Exactly like A [Logger](#), an [AppenderBuffer](#) can have an arbitrary amount of Appenders attached. When the buffer is flushed, the buffered events are dispatched to these Appenders.

*Usage:*

```
AppenderBuffer$set_appenders(x)
```

*Arguments:*

x single [Appender](#) or a list thereof. Appenders control the output of a Logger. Be aware that a Logger also inherits the Appenders of its ancestors (see vignette("lgr", package = "lgr") for more info about Logger inheritance).

**Method** add\_appender(): Add an Appender to the AppenderBuffer

Add or remove an [Appender](#). Supplying a name is optional but recommended. After adding an Appender with appender\$add\_appender(AppenderConsole\$new(), name = "console") you can refer to it via appender\$appenders\$console. remove\_appender() can remove an Appender by position or name.

*Usage:*

```
AppenderBuffer$add_appender(appender, name = NULL)
```

*Arguments:*

appender a single [Appender](#)

name a character scalar. Optional but recommended.

**Method** remove\_appender(): remove an appender

*Usage:*

```
AppenderBuffer$remove_appender(pos)
```

*Arguments:*

pos integer index or character name of the Appender(s) to remove

**Method** format():*Usage:*

```
AppenderBuffer$format(...)
```

**See Also**

[LayoutFormat](#)

Other Appenders: [AppenderConsole](#), [AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [AppenderFileRotating](#), [AppenderFile](#), [AppenderTable](#), [Appender](#)

---

AppenderConsole	<i>Log to the console</i>
-----------------	---------------------------

---

**Description**

An Appender that outputs to the R console. If you have the package **crayon** installed log levels will be coloured by default (but you can modify this behaviour by passing a custom [Layout](#)).

**Super classes**

[lgr::Filterable](#) -> [lgr::Appender](#) -> AppenderConsole

**Methods****Public methods:**

- [AppenderConsole\\$new\(\)](#)
- [AppenderConsole\\$append\(\)](#)

**Method** new():*Usage:*

```
AppenderConsole$new(
  threshold = NA_integer_,
  layout = LayoutFormat$new(fmt = "%L [%t] %m %f", timestamp_fmt = "%H:%M:%OS3",
    colors = getOption("lgr.colors", list())),
  filters = NULL
)
```

**Method** append():*Usage:*

```
AppenderConsole$append(event)
```

**See Also**[LayoutFormat](#)Other Appenders: [AppenderBuffer](#), [AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [AppenderFileRotating](#), [AppenderFile](#), [AppenderTable](#), [Appender](#)**Examples**

```
# create a new logger with propagate = FALSE to prevent routing to the root
# logger. Please look at the section "Logger Hierarchies" in the package
# vignette for more info.
lg <- get_logger("test")$set_propagate(FALSE)

lg$add_appender(AppenderConsole$new())
lg$add_appender(AppenderConsole$new(
  layout = LayoutFormat$new("[%t] %c(): [%n] %m", colors = getOption("lgr.colors"))))

# Will output the message twice because we attached two console appenders
lg$warn("A test message")
lg$config(NULL) # reset config
```

---

**AppenderFile***Log to a file*

---

**Description**

A simple Appender that outputs to a file in the file system. If you plan to log to text files, consider logging to JSON files and take a look at [AppenderJson](#), which is a shortcut for AppenderFile preconfigured with [LayoutJson](#).

**Super classes**

```
lgr::Filterable -> lgr::Appender -> AppenderFile
```

**Active bindings**

file character scalar. path to the log file

data data.frame. Contents of file parsed to a data.frame if used with a [Layout](#) that supports parsing of log file data (notably [LayoutJson](#)). Will throw an error if Layout does not support parsing.

data character scalar. Like \$data, but returns a data.table instead (requires the **data.table** package).

**Methods****Public methods:**

- [AppenderFile\\$new\(\)](#)
- [AppenderFile\\$append\(\)](#)
- [AppenderFile\\$set\\_file\(\)](#)
- [AppenderFile\\$show\(\)](#)

**Method new():**

*Usage:*

```
AppenderFile$new(  
  file,  
  threshold = NA_integer_,  
  layout = LayoutFormat$new(),  
  filters = NULL  
)
```

**Method append():**

*Usage:*

```
AppenderFile$append(event)
```

**Method set\_file():** Set a log file

*Usage:*

```
AppenderFile$set_file(file)
```

*Arguments:*

file character scalar. Path to the log file. If file does not exist it will be created.

**Method show():** Display the contents of the log file.

*Usage:*

```
AppenderFile$show(threshold = NA_integer_, n = 20L)
```

*Arguments:*

threshold character or integer scalar. The minimum log level that should be displayed.

n integer scalar. Show only the last n log entries that match threshold.

**Super classes**

```
lgr::Filterable -> lgr::Appender -> lgr::AppenderFile -> AppenderJson
```

**Methods****Public methods:**

- [AppenderJson\\$new\(\)](#)

**Method new():**

*Usage:*

```

AppenderJson$new(
  file,
  threshold = NA_integer_,
  layout = LayoutJson$new(),
  filters = NULL
)

```

### See Also

[LayoutFormat](#), [LayoutJson](#)

[LayoutFormat](#), [LayoutJson](#)

Other Appenders: [AppenderBuffer](#), [AppenderConsole](#), [AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [AppenderFileRotating](#), [AppenderTable](#), [Appender](#)

Other Appenders: [AppenderBuffer](#), [AppenderConsole](#), [AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [AppenderFileRotating](#), [AppenderTable](#), [Appender](#)

### Examples

```

lg <- get_logger("test")
default <- tempfile()
fancy <- tempfile()
json <- tempfile()

lg$add_appender(AppenderFile$new(default), "default")
lg$add_appender(
  AppenderFile$new(fancy, layout = LayoutFormat$new("[%t] %c(): %L %m")), "fancy"
)
lg$add_appender(
  AppenderFile$new(json, layout = LayoutJson$new()), "json"
)

lg$info("A test message")

readLines(default)
readLines(fancy)
readLines(json)

# cleanup
lg$config(NULL)
unlink(default)
unlink(fancy)
unlink(json)
tf <- tempfile()
lg <- get_logger("test")$
  set_appenders(AppenderJson$new(tf))$
  set_propagate(FALSE)

lg$info("A test message")
lg$info("A test message %s strings", "with format strings", and = "custom_fields")

lg$appenders[[1]]$show()

```



```
lg$appenders[[1]]$data

# cleanup
lg$config(NULL)
unlink(tf)
```

---

AppenderFileRotating *Log to a rotating file*

---

## Description

Log to a rotating file

Log to a rotating file

## Details

An extension of [AppenderFile](#) that rotates logfiles based on certain conditions. Please refer to the documentation of [rotor::rotate\(\)](#) for the meanings of the extra arguments

## Super classes

[lgr::Filterable](#) -> [lgr::Appender](#) -> [lgr::AppenderFile](#) -> AppenderFileRotating

## Active bindings

`backups` A data.frame containing information on path, file size, etc... on the available backups of file.

## Methods

### Public methods:

- [AppenderFileRotating\\$new\(\)](#)
- [AppenderFileRotating\\$append\(\)](#)
- [AppenderFileRotating\\$rotate\(\)](#)
- [AppenderFileRotating\\$prune\(\)](#)
- [AppenderFileRotating\\$set\\_file\(\)](#)
- [AppenderFileRotating\\$set\\_size\(\)](#)
- [AppenderFileRotating\\$set\\_max\\_backups\(\)](#)
- [AppenderFileRotating\\$set\\_compression\(\)](#)
- [AppenderFileRotating\\$set\\_create\\_file\(\)](#)
- [AppenderFileRotating\\$set\\_backup\\_dir\(\)](#)
- [AppenderFileRotating\\$format\(\)](#)
- [AppenderFileRotating\\$clone\(\)](#)

**Method** `new()`:

*Usage:*

```
AppenderFileRotating$new(  
  file,  
  threshold = NA_integer_,  
  layout = LayoutFormat$new(),  
  filters = NULL,  
  size = Inf,  
  max_backups = Inf,  
  compression = FALSE,  
  backup_dir = dirname(file),  
  create_file = NULL  
)
```

*Arguments:*

size, max\_backups, compression, backup\_dir, fmt see [rotor::rotate\(\)](#) for the meaning of these arguments. Note that fmt corresponds to format and backup\_dir to dir.

**Method** append():*Usage:*

```
AppenderFileRotating$append(event)
```

**Method** rotate():*Usage:*

```
AppenderFileRotating$rotate(force = FALSE)
```

**Method** prune():*Usage:*

```
AppenderFileRotating$prune(max_backups = self$max_backups)
```

**Method** set\_file():*Usage:*

```
AppenderFileRotating$set_file(file)
```

**Method** set\_size():*Usage:*

```
AppenderFileRotating$set_size(x)
```

**Method** set\_max\_backups():*Usage:*

```
AppenderFileRotating$set_max_backups(x)
```

**Method** set\_compression():*Usage:*

```
AppenderFileRotating$set_compression(x)
```

**Method** set\_create\_file():*Usage:*

AppenderFileRotating\$set\_create\_file(x)

**Method** set\_backup\_dir():

*Usage:*

AppenderFileRotating\$set\_backup\_dir(x)

**Method** format():

*Usage:*

AppenderFileRotating\$format(color = false, ...)

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

AppenderFileRotating\$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

**See Also**

[AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [rotor::rotate\(\)](#)

Other Appenders: [AppenderBuffer](#), [AppenderConsole](#), [AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [AppenderFile](#), [AppenderTable](#), [Appender](#)

---

AppenderFileRotatingDate

*Log to a date-stamped rotating file*

---

**Description**

Log to a date-stamped rotating file

Log to a date-stamped rotating file

**Details**

This is a simpler version of [AppenderFileRotatingTime](#) when the timestamps do not need to include sub-day accuracy.

**Super classes**

[lgr::Filterable](#) -> [lgr::Appender](#) -> [lgr::AppenderFile](#) -> [lgr::AppenderFileRotating](#)  
-> [AppenderFileRotatingDate](#)

## Methods

### Public methods:

- [AppenderFileRotatingDate\\$new\(\)](#)
- [AppenderFileRotatingDate\\$clone\(\)](#)

### Method `new()`:

*Usage:*

```
AppenderFileRotatingDate$new(  
  file,  
  threshold = NA_integer_,  
  layout = LayoutFormat$new(),  
  filters = NULL,  
  age = Inf,  
  size = -1,  
  max_backups = Inf,  
  compression = FALSE,  
  backup_dir = dirname(file),  
  fmt = "%Y-%m-%d",  
  overwrite = FALSE,  
  cache_backups = TRUE,  
  create_file = NULL  
)
```

*Arguments:*

size, age, max\_backups, compression, backup\_dir, fmt, overwrite, cache\_backups see [rotor::rotate\\_date\(\)](#) for the meaning of these arguments. Note that `fmt` corresponds to `format` (because `$format` has a special meaning for R6 classes).

**Method `clone()`:** The objects of this class are cloneable with this method.

*Usage:*

```
AppenderFileRotatingDate$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

[AppenderFileRotatingTime](#), [AppenderFileRotating](#), [rotor::rotate\(\)](#)

Other Appenders: [AppenderBuffer](#), [AppenderConsole](#), [AppenderFileRotatingTime](#), [AppenderFileRotating](#), [AppenderFile](#), [AppenderTable](#), [Appender](#)

---

AppenderFileRotatingTime

*Log to a time-stamped rotating file*

---

## Description

Log to a time-stamped rotating file

Log to a time-stamped rotating file

## Super classes

```
lgr::Filterable -> lgr::Appender -> lgr::AppenderFile -> lgr::AppenderFileRotating  
-> AppenderFileRotating
```

## Active bindings

cache\_backups TRUE or FALSE. If TRUE (the default) the list of backups is cached, if FALSE it is read from disk every time this appender triggers. Caching brings a significant speedup for checking whether to rotate or not based on the age of the last backup, but is only safe if there are no other programs/functions (except this appender) interacting with the backups.

## Methods

### Public methods:

- `AppenderFileRotatingTime$new()`
- `AppenderFileRotatingTime$rotate()`
- `AppenderFileRotatingTime$set_age()`
- `AppenderFileRotatingTime$set_fmt()`
- `AppenderFileRotatingTime$set_overwrite()`
- `AppenderFileRotatingTime$set_cache_backups()`
- `AppenderFileRotatingTime$format()`
- `AppenderFileRotatingTime$clone()`

### Method `new()`:

*Usage:*

```
AppenderFileRotatingTime$new(  
  file,  
  threshold = NA_integer_,  
  layout = LayoutFormat$new(),  
  filters = NULL,  
  age = Inf,  
  size = -1,  
  max_backups = Inf,  
  compression = FALSE,  
  backup_dir = dirname(file),
```

```

    fmt = "%Y-%m-%d--%H-%M-%S",
    overwrite = FALSE,
    cache_backups = TRUE,
    create_file = NULL
)

```

**Arguments:**

size, age, max\_backups, compression, backup\_dir, fmt, overwrite, cache\_backups see [rotor::rotate\\_time\(\)](#) for the meaning of these arguments. Note that fmt corresponds to format and backup\_dir to dir.

**Method rotate():***Usage:*

```
AppenderFileRotatingTime$rotate(force = FALSE, now = Sys.time())
```

**Method set\_age():***Usage:*

```
AppenderFileRotatingTime$set_age(x)
```

**Method set\_fmt():***Usage:*

```
AppenderFileRotatingTime$set_fmt(x)
```

**Method set\_overwrite():***Usage:*

```
AppenderFileRotatingTime$set_overwrite(x)
```

**Method set\_cache\_backups():** set the cache\_backups flag.*Usage:*

```
AppenderFileRotatingTime$set_cache_backups(x)
```

*Arguments:*

x a logical scalar

**Method format():***Usage:*

```
AppenderFileRotatingTime$format(color = FALSE, ...)
```

**Method clone():** The objects of this class are cloneable with this method.*Usage:*

```
AppenderFileRotatingTime$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

**See Also**

[AppenderFileRotatingDate](#), [AppenderFileRotating](#), [rotor::rotate\(\)](#)

Other Appenders: [AppenderBuffer](#), [AppenderConsole](#), [AppenderFileRotatingDate](#), [AppenderFileRotating](#), [AppenderFile](#), [AppenderTable](#), [Appender](#)

---

AppenderMemory

*Abstract class for logging to memory buffers*


---

## Description

**NOTE:** This is an *abstract class*. Abstract classes cannot be instantiated directly, but are exported for package developers that want to extend `lgr` - for example by creating their own [Appenders](#) or [Layouts](#). Please refer to the *see also* section for actual implementations of this class.

AppenderMemory is extended by Appenders that retain an in-memory event buffer, such as [AppenderBuffer](#) and [AppenderPushbullet](#) from the `lgrExtra` package.

## Super classes

`lgr::Filterable` -> `lgr::Appender` -> AppenderMemory

## Active bindings

`flush_on_exit` A logical scalar. Should the buffer be flushed if the Appender is destroyed (e.g. because the R session is terminated)?

`flush_on_rotate` A logical scalar. Should the buffer be flushed when it is rotated because `$buffer_size` is exceeded?

`should_flush` A function with exactly one arguments: `event`. `$append()` calls this function internally on the current [LogEvent](#) and flushes the buffer if it evaluates to TRUE.

`buffer_size` integer scalar  $\geq 0$ . Maximum number of [LogEvents](#) to buffer.

`flush_threshold` A numeric or character threshold. [LogEvents](#) with a `log_level` equal to or lower than this threshold trigger flushing the buffer.

`buffer_events` A list of [LogEvents](#). Contents of the buffer.

`buffer_events` A `data.frame`. Contents of the buffer converted to a `data.frame`.

`buffer_events` A `data.frame`. Contents of the buffer converted to a `data.table`.

## Methods

### Public methods:

- `AppenderMemory$new()`
- `AppenderMemory$append()`
- `AppenderMemory$flush()`
- `AppenderMemory$clear()`
- `AppenderMemory$set_buffer_size()`
- `AppenderMemory$set_should_flush()`
- `AppenderMemory$set_flush_on_exit()`
- `AppenderMemory$set_flush_on_rotate()`
- `AppenderMemory$set_flush_threshold()`
- `AppenderMemory$show()`

- [AppenderMemory\\$format\(\)](#)

**Method** new():

*Usage:*

AppenderMemory\$new(...)

**Method** append():

*Usage:*

AppenderMemory\$append(event)

**Method** flush(): Sends the buffer's contents to all attached Appenders and then clears the Buffer

*Usage:*

AppenderMemory\$flush()

**Method** clear(): Clears the buffer, discarding all buffered Events

*Usage:*

AppenderMemory\$clear()

**Method** set\_buffer\_size(): Set the maximum size of the buffer

*Usage:*

AppenderMemory\$set\_buffer\_size(x)

*Arguments:*

x an integer scalar  $\geq 0$ . Number of [LogEvents](#) to buffer.

**Method** set\_should\_flush(): Set function that can trigger flushing the buffer

*Usage:*

AppenderMemory\$set\_should\_flush(x)

*Arguments:*

x A function with the single argument event. Setting x to NULL is a shortcut for function(event) FALSE. See active bindings.

**Method** set\_flush\_on\_exit(): Should the buffer be flushed when the Appender is destroyed?

*Usage:*

AppenderMemory\$set\_flush\_on\_exit(x)

*Arguments:*

x A logical scalar. See active bindings.

**Method** set\_flush\_on\_rotate(): Should the buffer be flushed if buffer\_size is exceeded?

*Usage:*

AppenderMemory\$set\_flush\_on\_rotate(x)

*Arguments:*

x A logical scalar. See active bindings.



**Method** `set_flush_threshold()`: Set threshold that triggers flushing

*Usage:*

```
AppenderMemory$set_flush_threshold(level)
```

*Arguments:*

`level` A numeric or character [threshold](#). See active bindings.

**Method** `show()`: Display the contents of the log table. Relies on the `$format_event` method of the [Layout](#) attached to this Appender.

*Usage:*

```
AppenderMemory$show(threshold = NA_integer_, n = 20L)
```

*Arguments:*

`threshold` character or integer scalar. The minimum log level that should be displayed.

`n` integer scalar. Show only the last `n` log entries that match `threshold`.

**Method** `format()`:

*Usage:*

```
AppenderMemory$format(color = FALSE, ...)
```

## See Also

[LayoutFormat](#)

Other abstract classes: [AppenderTable](#), [Appender](#), [Filterable](#)

AppenderTable

*Abstract class for logging to tabular structures*

## Description

**NOTE:** This is an *abstract class*. Abstract classes cannot be instantiated directly, but are exported for package developers that want to extend `lgr` - for example by creating their own [Appenders](#) or [Layouts](#). Please refer to the *see also* section for actual implementations of this class.

`AppenderTable` is extended by `Appenders` that write to a data source that can be interpreted as tables, (usually a `data.frame`). Examples are `AppenderDbi`, `AppenderRjdbc` and `AppenderDt` from the [lgrExtra](#) package.

## Super classes

```
lgr::Filterable -> lgr::Appender -> AppenderTable
```

## Active bindings

`data` character scalar. Contents of the table, parsed to a `data.frame`.

`data` character scalar. Like `$data`, but returns a `data.table` instead (requires the **data.table** package).

**Methods****Public methods:**

- [AppenderTable\\$new\(\)](#)
- [AppenderTable\\$show\(\)](#)
- [AppenderTable\\$format\(\)](#)

**Method new():***Usage:*

AppenderTable\$new(...)

**Method show():** Show recent log entries*Usage:*

AppenderTable\$show(threshold = NA\_integer\_, n = 20L)

*Arguments:*

threshold an integer or character [threshold](#). Only show events with a log level at or below this threshold.

n a positive integer scalar. Show at most that many entries

**Method format():***Usage:*

AppenderTable\$format(color = FALSE, ...)

**See Also**

Other abstract classes: [AppenderMemory](#), [Appender](#), [Filterable](#)

Other Appenders: [AppenderBuffer](#), [AppenderConsole](#), [AppenderFileRotatingDate](#), [AppenderFileRotatingTime](#), [AppenderFileRotating](#), [AppenderFile](#), [Appender](#)

---

 as.data.frame.LogEvent

*Coerce LogEvents to Data Frames*

---

**Description**

Coerce LogEvents to data.frames, [data.tables](#), or [tibbles](#).

**Usage**

```
## S3 method for class 'LogEvent'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  stringsAsFactors = FALSE,
```

```

    ...,
    box_if = function(.) !(is.atomic(.) && identical(length(.), 1L)),
    cols_expand = NULL
  )

as.data.table.LogEvent(
  x,
  ...,
  box_if = function(.) !(is.atomic(.) && identical(length(.), 1L)),
  cols_expand = "msg"
)

as_tibble.LogEvent(
  x,
  ...,
  box_if = function(.) !(is.atomic(.) && identical(length(.), 1L)),
  cols_expand = "msg"
)

```

### Arguments

x	any R object.
row.names	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
optional	currently ignored and only included for compatibility.
stringsAsFactors	logical scalar: should character vectors be converted to factors? Defaults to FALSE (as opposed to <code>base::as.data.frame()</code> ) and is only included for compatibility.
...	passed on to <code>data.frame()</code>
box_if	a function that returns TRUE or FALSE to determine which values are to be boxed (i.e. placed as single elements in a list column). See example
cols_expand	character vector. Columns to <i>not</i> box (even if <code>box_if()</code> returns TRUE). Vectors in these columns will result in multiple rows in the result (rather than a single list-column row). This defaults to "msg" for vectorized logging over the log message.

### See Also

[data.table::data.table](#), [tibble::tibble](#)

### Examples

```

lg <- get_logger("test")
lg$info("lorem ipsum")
as.data.frame(lg$last_event)

lg$info("LogEvents can store any custom log values", df = iris)

```

```

as.data.frame(lg$last_event)
head(as.data.frame(lg$last_event)$df[[1]])

# how boxing works

# by default non-scalars are boxed
lg$info("letters", letters = letters)
as.data.frame(lg$last_event)

# this behaviour can be modified by supplying a custom boxing function
as.data.frame(lg$last_event, box_if = function(.) FALSE)
as.data.frame(lg$last_event, cols_expand = "letters")

# The `msg` argument of a log event is always vectorized
lg$info(c("a vectorized", "log message"))
as.data.frame(lg$last_event)

lg$config(NULL)

```

---

as\_LogEvent

*Coerce objects to LogEvent*


---

## Description

Smartly coerce R objects that look like LogEvents to LogEvents. Mainly useful for developing Appenders.

## Usage

```

as_LogEvent(x, ...)

## S3 method for class 'list'
as_LogEvent(x, ...)

## S3 method for class 'data.frame'
as_LogEvent(x, ...)

```

## Arguments

x	any supported R object
...	currently ignored

## Details

**Note:** `as_LogEvent.data.frame()` only supports single-row data.frames

## Value

a [LogEvent](#)

**See Also**

Other docs relevant for extending lgr: [LogEvent](#), [event\\_list\(\)](#), [standardize\\_threshold\(\)](#)

---

 basic\_config

*Basic Setup for the Logging System*


---

**Description**

A quick and easy way to configure the root logger. This is less powerful than using `lgr$config()` or `lgr$set_*` (see [Logger](#)), but reduces the most common configurations to a single line of code.

**Usage**

```
basic_config(
  file = NULL,
  fmt = "%L [%t] %m",
  timestamp_fmt = "%Y-%m-%d %H:%M:%OS3",
  threshold = "info",
  appenders = NULL,
  console = if (is.null(appenders)) "all" else FALSE,
  console_fmt = "%L [%t] %m %f",
  console_timestamp_fmt = "%H:%M:%OS3",
  memory = FALSE
)
```

**Arguments**

file	character scalar: If not NULL a <a href="#">AppenderFile</a> will be created that logs to this file. If the filename ends in <code>.jsonl</code> , the Appender will be set up to use the <a href="#">JSON Lines</a> format instead of plain text (see <a href="#">AppenderFile</a> and <a href="#">AppenderJson</a> ).
fmt	character scalar: Format to use if file is supplied and not a <code>.jsonl</code> file. If NULL it defaults to <code>"%L [%t] %m"</code> (see <a href="#">format.LogEvent</a> )
timestamp_fmt	see <a href="#">format.POSIXct()</a>
threshold	character or integer scalar. The minimum <a href="#">log level</a> that should be processed by the root logger.
appenders	a single <a href="#">Appender</a> or a list thereof.
console	logical scalar or a threshold (see above). Add an appender logs to the console (i.e. displays messages in an interactive R session)
console_fmt	character scalar: like <code>fmt</code> but used for console output
console_timestamp_fmt	character scalar: like <code>timestamp_fmt</code> but used for console output
memory	logical scalar. or a threshold (see above). Add an Appender that logs to a memory buffer, see also <a href="#">show_log()</a> and <a href="#">AppenderBuffer</a>

**Value**

the root Logger (lgr)

**Examples**

```
# log to a file
basic_config(file = tempfile())
unlink(lgr$appenders$file$file) # cleanup

basic_config(file = tempfile(fileext = "jsonl"))
unlink(lgr$appenders$file$file) # cleanup

# log debug messages to a memory buffer
basic_config(threshold = "all", memory = "all", console = "info")
lgr$info("an info message")
lgr$debug("a hidden message")
show_log()

# reset to default config
basic_config()
```

---

CannotInitializeAbstractClassError  
*Logger Error Conditions*

---

**Description**

Logger Error Conditions

**Usage**

```
CannotInitializeAbstractClassError(class = parent.frame(2)[["classes"]])
```

**Arguments**

class	character scalar. The abstract class that was mistakenly tried to initialize. The default is to discover the class name automatically if called inside <code>\$initialize(){...}</code> in an <a href="#">R6::R6</a> class definition
-------	---

**Value**

a condition object

---

colorize_levels	<i>Colorize Levels</i>
-----------------	------------------------

---

## Description

Colorize Levels

## Usage

```
colorize_levels(  
  x,  
  colors = getOption("lgr.colors", NULL),  
  transform = identity  
)
```

## Arguments

x	numeric or character levels to be colored. Unlike in many other functions in lgr, character levels are <i>not</i> case sensitive in this function and leading/trailing whitespace is ignored to make it more comfortable to use <code>colorize_levels()</code> inside formatting functions.
colors	A list of functions that will be used to color the log levels (likely from <a href="#">crayon::crayon</a> ).
transform	a function to transform x (for example <code>toupper()</code> )

## Value

a character vector with color ANSI codes

## See Also

Other formatting utils: [label\\_levels\(\)](#)

## Examples

```
cat(colorize_levels(c(100, 200)))  
cat(colorize_levels(c("trace", "warn ", "DEBUG")))  
cat(colorize_levels(c("trace", "warn ", "DEBUG"), transform = function(x) strtrim(x, 1) ))
```

---

```
default_exception_handler
```

*Demote an exception to a warning*

---

### Description

Throws a timestamped warning instead of stopping the program. This is the default exception handler used by [Loggers](#).

### Usage

```
default_exception_handler(e)
```

### Arguments

e                    an error condition object

### Value

The warning as character vector

### Examples

```
tryCatch(stop("an error has occurred"), error = default_exception_handler)
```

---

```
EventFilter
```

*Event Filters*

---

### Description

EventFilters specify arbitrarily complex logic for whether or not a LogEvent should be processed by a [Logger](#) or [Appender](#). They are attached to Loggers/Appenders via their `$set_filter()` or `$add_filter()` methods. If any EventFilter evaluates to FALSE for a given event, that event is ignored - similarly to when it does not pass the objects' threshold.

Usually you do not need to instantiate a formal EventFilter object as you can just use any function that has the single argument event instead. If you need to implement more complex filter logic - for example a filter that is dependent on a dataset - it might be desirable to subclass EventFilter, as [R6::R6](#) objects can store data and functions together.

`.obj()` is a special function that can only be used within the `$filter()` methods of [EventFilters](#). It returns the [Logger](#) or [Appender](#) that the EventFilter is attached to.

### Usage

```
.obj()
```



## Modifying LogEvents with EventFilters

Since LogEvents are R6 objects with reference semantics, EventFilters can be abused to modify events before passing them on. lgr comes with a few preset filters that use this property: [FilterInject](#) (similar to [with\\_log\\_level\(\)](#)) and [FilterForceLevel](#) (similar to [with\\_log\\_value\(\)](#)).

**NOTE:** The base class for Filters is called `EventFilter` so that it doesn't conflict with `base::Filter()`. The recommended convention for Filter subclasses is to call them `FilterSomething` and leave out the Event prefix.

## Methods

### Public methods:

- [EventFilter\\$new\(\)](#)
- [EventFilter\\$clone\(\)](#)

**Method** `new()`: Initialize a new EventFilter

*Usage:*

```
EventFilter$new(fun = function(event) TRUE)
```

*Arguments:*

fun a function with a single argument event that must return either TRUE or FALSE. Any non-FALSE will be interpreted as TRUE (= no filtering takes place) and a warning will be thrown.

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
EventFilter$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

[is\\_filter\(\)](#)

## Examples

```
lg <- get_logger("test")
f <- function(event) {
  cat("via event$.logger:", event$.logger$threshold, "\n") # works for loggers only
  cat("via .obj():      ", .obj()$threshold, "\n") # works for loggers and appenders
  TRUE
}
lg$add_filter(f)
lg$fatal("test")
lg$config(NULL)
```

---

event_list	<i>A List of LogEvents</i>
------------	----------------------------

---

### Description

An `event_list` is a class for `list()`s whose only elements are [LogEvents](#). This structure is occasionally used internally in `lgr` (for example by [AppenderBuffer](#)) and can be useful for developers that want to write their own `Appenders`.

### Usage

```
event_list(...)

as_event_list(x, ...)

## S3 method for class 'list'
as_event_list(x, ..., scalarize = FALSE)

## S3 method for class 'LogEvent'
as_event_list(x, ..., scalarize = FALSE)

## S3 method for class 'data.frame'
as_event_list(x, na.rm = TRUE, ...)

as.data.table.event_list(x, na.rm = TRUE)

## S3 method for class 'event_list'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  stringsAsFactors = FALSE,
  na.rm = TRUE,
  ...
)
```

### Arguments

<code>...</code>	for event elements to be added to the list, for the <code>as_*</code> () functions parameters passed on to methods.
<code>x</code>	any R object
<code>scalarize</code>	logical scalar. Turn <a href="#">LogEvents</a> with non-scalar <code>msg</code> field into separate log events
<code>na.rm</code>	remove NA values before coercing a <code>data.frame</code> to an <code>event_list()</code> .
<code>row.names</code>	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.

optional currently ignored and only included for compatibility.  
 stringsAsFactors logical scalar: should character vectors be converted to factors? Defaults to FALSE (as opposed to `base::as.data.frame()`) and is only included for compatibility.

### Details

For convenience, `as.data.frame()` and `as.data.table()` methods exist for event lists.

### Value

`as_event_list()` and `as_event_list()` return a flat list of [LogEvents](#). Nested lists get automatically flattened.

`as.data.frame` and `as.data.table` return a `data.frame` or `data.table` respectively

### See Also

Other docs relevant for extending `lgr`: [LogEvent](#), [as\\_LogEvent\(\)](#), [standardize\\_threshold\(\)](#)

### Examples

```
e <- LogEvent$new(level = 300, msg = "a", logger = lgr)
as_event_list(e)
as_event_list(c(e, e))
# nested lists get automatically unnested
as_event_list(c(e, list(nested_event = e)))

# scalarize = TRUE "unpacks" events with vector log messages
e <- LogEvent$new(level = 300, msg = c("A", "B"), logger = lgr)
as_event_list(e, scalarize = FALSE)
as_event_list(e, scalarize = TRUE)
```

---

 Filterable

*Abstract Class for Filterables*


---

### Description

Superclass for classes that have a `$filter()` method such as [Appenders](#) and [Loggers](#). See [Event-Filter](#) for details.

**NOTE:** This is an *abstract class*. Abstract classes cannot be instantiated directly, but are exported for package developers that want to extend `lgr` - for example by creating their own [Appenders](#) or [Layouts](#). Please refer to the *see also* section for actual implementations of this class.

### Active bindings

`filters` a list of all attached Filters.

**Methods****Public methods:**

- [Filterable\\$filter\(\)](#)
- [Filterable\\$add\\_filter\(\)](#)
- [Filterable\\$remove\\_filter\(\)](#)
- [Filterable\\$set\\_filters\(\)](#)

**Method** [filter\(\)](#): Determine whether the LogEvent x should be passed on to Appenders (TRUE) or not (FALSE). See also the active binding filters.

*Usage:*

```
Filterable$filter(event)
```

*Arguments:*

event a [LogEvent](#)

**Method** [add\\_filter\(\)](#): Attach a filter

*Usage:*

```
Filterable$add_filter(filter, name = NULL)
```

*Arguments:*

filter • a function with the single argument event that returns TRUE or FALSE;

- an [EventFilter R6::R6](#) object; or
- any R object with a `$filter()` method.

If a Filter returns a non-FALSE value, will be interpreted as TRUE (= no filtering takes place) and a warning will be thrown.

name character scalar or NULL. An optional name which makes it easier to access (or remove) the filter

**Method** [remove\\_filter\(\)](#): Remove a filter

*Usage:*

```
Filterable$remove_filter(pos)
```

*Arguments:*

pos character or integer scalar. The name or index of the Filter to be removed.

**Method** [set\\_filters\(\)](#): Set or replace (all) Filters of parent object. See [EventFilter](#) for how Filters work.

*Usage:*

```
Filterable$set_filters(filters)
```

*Arguments:*

filters a list (named or unnamed) of [EventFilters](#) or predicate functions. See [is\\_filter\(\)](#).

**See Also**

Other abstract classes: [AppenderMemory](#), [AppenderTable](#), [Appender](#)

---

FilterForceLevel	<i>Override the log level of all events processed by a Logger/Appender</i>
------------------	--

---

### Description

Overrides the log level of the Appender/Logger that this filter is attached to to with level. See also [with\\_log\\_level\(\)](#). It is recommended to use filters that modify LogEvents only with Loggers, but they will also work with Appenders.

### Super class

```
lgr::EventFilter -> FilterForceLevel
```

### Public fields

level an integer [log level](#) used to override the log levels of each [LogEvent](#) processed by this filter.

### Methods

#### Public methods:

- [FilterForceLevel\\$new\(\)](#)
- [FilterForceLevel\\$clone\(\)](#)

**Method** [new\(\)](#): Initialize a new FilterForceLevel

*Usage:*

```
FilterForceLevel$new(level)
```

*Arguments:*

level an integer or character [log level](#)

**Method** [clone\(\)](#): The objects of this class are cloneable with this method.

*Usage:*

```
FilterForceLevel$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

### Examples

```
lg <- get_logger("test")

analyse <- function(){
  lg$add_filter(FilterForceLevel$new("info"), "force")
  on.exit(lg$remove_filter("force"))
  lg$error("an error with forced log level INFO")
}
```

```
analyse()
lg$error("an normal error")
lg$config(NULL) # reset config
```

---

 FilterInject

*Inject values into all events processed by a Logger/Appender*


---

### Description

Inject arbitrary values into all [LogEvents](#) processed by a Logger/Appender. It is recommended to use filters that modify LogEvents only with Loggers, but they will also work with Appenders.

### Super class

`lgr::EventFilter` -> FilterInject

### Public fields

`values` a named list of values to be injected into each [LogEvent](#) processed by this filter

### Methods

#### Public methods:

- [FilterInject\\$new\(\)](#)
- [FilterInject\\$clone\(\)](#)

**Method** `new()`: Initialize a new FilterInject

*Usage:*

```
FilterInject$new(..., .list = list())
```

*Arguments:*

`...`, `.list` any number of named R objects that will be injected as custom fields into all [LogEvents](#) processed by the Appender/Logger that this filter is attached to. See also [with\\_log\\_value\(\)](#).

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
FilterInject$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

**Examples**

```
lg <- get_logger("test")

analyse <- function(){
  lg$add_filter(FilterInject$new(type = "analysis"), "inject")
  on.exit(lg$remove_filter("inject"))
  lg$error("an error with forced custom 'type'-field")
}

analyse()
lg$error("an normal error")
lg$config(NULL) # reset config
```

---

get\_caller

*Information About the System*

---

**Description**

get\_caller() Tries to determine the calling functions based on where.

**Usage**

```
get_caller(where = -1L)

get_user(fallback = "unknown user")
```

**Arguments**

where            integer scalar (usually negative). Look up that many frames up the call stack  
fallback        A fallback in case the user name could not be determined

**Value**

a character scalar.

**See Also**

[base::sys.call\(\)](#)  
[whoami::whoami\(\)](#)

**Examples**

```
foo <- function() get_caller(-1L)
foo()
get_user()
```

---

get\_logger                      *Get/Create a Logger*

---

### Description

Get/Create a Logger

### Usage

```
get_logger(name, class = Logger, reset = FALSE)
```

```
get_logger_glue(name)
```

### Arguments

name	a character scalar or vector: The qualified name of the Logger as a hierarchical value.
class	An <a href="#">R6ClassGenerator</a> object. Usually Logger or LoggerGlue are the only valid choices.
reset	a logical scalar. If TRUE the logger is reset to an unconfigured state. Unlike <code>\$config(NULL)</code> this also replaces a LoggerGlue with vanilla Logger. Please note that this will invalidate Logger references created before the reset call (see examples).

### Value

a [Logger](#)

### Examples

```
lg <- get_logger("log/ger/test")
# equivalent to
lg <- get_logger(c("log", "ger", "test"))
lg$warn("a %s message", "warning")
lg
lg$parent

if (requireNamespace('glue')){
  lg <- get_logger_glue("log/ger")
}
lg$warn("a {.text} message", .text = "warning")

# completely reset 'glue' to an unconfigured vanilla Logger
get_logger("log/ger", reset = TRUE)
# WARNING: this invalidates existing references to the Logger
try(lg$info("lg has been invalidated an no longer works"))

lg <- get_logger("log/ger")
lg$info("now all is well again")
```



---

get\_log\_levels            *Manage Log Levels*

---

### Description

Display, add and remove character labels for log levels.

### Usage

```
get_log_levels()
add_log_levels(levels)
remove_log_levels(level_names)
```

### Arguments

levels            a named character vector (see examples)  
level\_names      a character vector of the names of the levels to remove

### Value

a named character vector of the globally available log levels (add\_log\_levels() and remove\_log\_levels() return invisibly).

### Default Log Levels

lgr comes with the following predefined log levels that are identical to the log levels of log4j.

Level	Name	Description
0	off	A log level of 0/off tells a Logger or Appender to suspend all logging
100	fatal	Critical error that leads to program abort. Should always indicate a stop() or similar
200	error	A severe error that does not trigger program abort
300	warn	A potentially harmful situation, like warning()
400	info	An informational message on the progress of the application
500	debug	Finer grained informational messages that are mostly useful for debugging
600	trace	An even finer grained message than debug
NA	all	A log level of NA/all tells a Logger or Appender to process all log events

### Examples

```
get_log_levels()
add_log_levels(c(errorish = 250))
get_log_levels()
remove_log_levels("errorish")
get_log_levels()
```

---

is_filter	<i>Check if an R Object is a Filter</i>
-----------	---

---

**Description**

Returns TRUE for any R object that can be used as a Filter for [Loggers](#) or [Appenders](#):

- a function with the single argument event;
- an [EventFilter R6::R6](#) object; or
- any object with a `$filter(event)` method.

**Note:** A Filter **must** return a scalar TRUE or FALSE, but this property cannot be checked by `is_filter()`.

**Usage**

```
is_filter(x)
```

**Arguments**

x	any R Object
---	--------------

**Value**

TRUE or FALSE

**See Also**

[EventFilter](#), [Filterable](#)

---

label_levels	<i>Label/Unlabel Log Levels</i>
--------------	---------------------------------

---

**Description**

Label/Unlabel Log Levels

**Usage**

```
label_levels(levels, log_levels = getOption("lgr.log_levels"))
```

```
unlabel_levels(labels, log_levels = getOption("lgr.log_levels"))
```

**Arguments**

levels	an integer vector of log levels
log_levels	named integer vector of valid log levels
labels	a character vector of log level labels. Please note that log levels are lowercase by default, even if many appenders print them in uppercase.

**Value**

a character vector for `label_levels()` and an integer vector for `unlabel_levels`

**See Also**

[get\\_log\\_levels\(\)](#)

Other formatting utils: [colorize\\_levels\(\)](#)

**Examples**

```
x <- label_levels(c(seq(0, 600, by = 100), NA))
print(x)
unlabel_levels(x)
```

---

Layout

*Abstract Class for Layouts*


---

**Description**

Abstract Class for Layouts

Abstract Class for Layouts

**Details**

[Appenders](#) pass [LogEvents](#) to a Layout which formats it for output. For the Layouts included in `lgr` that means turning the `LogEvent` into a character string.

For each Appender exist one more more possible Layouts, but not every Layout will work with every Appender. See the package `lgrExtra` for examples for Layouts that return different data types (such as `data.frames`) and Appenders that can handle them.

**Notes for developers**

Layouts may have an additional `$read(file, threshold, n)` method that returns a character vector, and/or an `$parse(file)` method that returns a `data.frame`. These can be used by Appenders to `$show()` methods and `$data` active bindings respectively (see source code of [AppenderFile](#)).

## Methods

### Public methods:

- [Layout\\$format\\_event\(\)](#)
- [Layout\\$toString\(\)](#)
- [Layout\\$clone\(\)](#)

**Method** `format_event()`: Format a log event

Function that the Layout uses to transform a [LogEvent](#) into something that an [Appender](#) can write to an output destination.

*Usage:*

```
Layout$format_event(event)
```

*Arguments:*

event a [LogEvent](#)

**Method** `toString()`:

*Usage:*

```
Layout$toString()
```

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
Layout$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

Other Layouts: [LayoutFormat](#), [LayoutGlue](#), [LayoutJson](#)

---

LayoutFormat

*Format Log Events as Text*

---

## Description

Format Log Events as Text

Format Log Events as Text

## Details

Format a [LogEvent](#) as human readable text using `format.LogEvent()`, which provides a quick and easy way to customize log messages. If you need more control and flexibility, consider using [LayoutGlue](#) instead.

see Fields

see Fields

see Fields

see Fields

Convert Layout to a character string Read a log file written using LayoutFormat

## Format Tokens

This is the same list of format tokens as for `format.LogEvent()`

`%t` The timestamp of the message, formatted according to `timestamp_fmt`)

`%l` the log level, lowercase character representation

`%L` the log level, uppercase character representation

`%k` the log level, first letter of lowercase character representation

`%K` the log level, first letter of uppercase character representation

`%n` the log level, integer representation

`%g` the name of the logger

`%p` the PID (process ID). Useful when logging code that uses multiple threads.

`%c` the calling function

`%m` the log message

`%f` all custom fields of `x` in a pseudo-JSON like format that is optimized for human readability and console output

`%j` all custom fields of `x` in proper JSON. This requires that you have **jsonlite** installed and does not support colors as opposed to `%f`

## Super class

`lgr::Layout` -> `LayoutFormat`

## Active bindings

`fmt` a character scalar containing format tokens. See `format.LogEvent()`.

`timestamp_fmt` a character scalar. See `base::format.POSIXct()`.

`colors` a named list of functions (like the ones provided by the package **crayon**) passed on on `format.LogEvent()`.

`pad_levels` "right", "left" or NULL. See `format.LogEvent()`.

## Methods

### Public methods:

- `LayoutFormat$new()`
- `LayoutFormat$format_event()`
- `LayoutFormat$set_fmt()`
- `LayoutFormat$set_timestamp_fmt()`
- `LayoutFormat$set_colors()`
- `LayoutFormat$set_pad_levels()`
- `LayoutFormat$toString()`
- `LayoutFormat$read()`
- `LayoutFormat$clone()`

**Method new():***Usage:*

```
LayoutFormat$new(  
  fmt = "%L [%t] %m %j",  
  timestamp_fmt = "%Y-%m-%d %H:%M:%OS3",  
  colors = NULL,  
  pad_levels = "right"  
)
```

**Method format\_event():** Format a LogEvent*Usage:*

```
LayoutFormat$format_event(event)
```

*Arguments:*event a [LogEvent](#)**Method set\_fmt():***Usage:*

```
LayoutFormat$set_fmt(x)
```

**Method set\_timestamp\_fmt():***Usage:*

```
LayoutFormat$set_timestamp_fmt(x)
```

**Method set\_colors():***Usage:*

```
LayoutFormat$set_colors(x)
```

**Method set\_pad\_levels():***Usage:*

```
LayoutFormat$set_pad_levels(x)
```

**Method toString():***Usage:*

```
LayoutFormat$toString()
```

**Method read():***Usage:*

```
LayoutFormat$read(file, threshold = NA_integer_, n = 20L)
```

*Arguments:*

threshold a character or integer threshold

n number of log entries to display

**Method clone():** The objects of this class are cloneable with this method.*Usage:*

```
LayoutFormat$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

**See Also**

Other Layouts: [LayoutGlue](#), [LayoutJson](#), [Layout](#)

**Examples**

```
# setup a dummy LogEvent
event <- LogEvent$new(
  logger = Logger$new("dummy logger"),
  level = 200,
  timestamp = Sys.time(),
  caller = NA_character_,
  msg = "a test message"
)
lo <- LayoutFormat$new()
lo$format_event(event)
```

---

LayoutGlue

*Format Log Events as Text via glue*

---

**Description**

Format a [LogEvent](#) as human readable text using `glue::glue`. The function is evaluated in an environment in which it has access to all elements of the [LogEvent](#) (see examples). This is more flexible than [LayoutFormat](#), but also more complex and slightly less performant.

**Super class**

`lgr::Layout` -> `LayoutGlue`

**Active bindings**

`fmt` A string that will be interpreted by `glue::glue()`

**Methods****Public methods:**

- `LayoutGlue$new()`
- `LayoutGlue$format_event()`
- `LayoutGlue$set_fmt()`
- `LayoutGlue$set_colors()`
- `LayoutGlue$toString()`
- `LayoutGlue$clone()`

**Method new():**

*Usage:*

```
LayoutGlue$new(
  fmt = "{pad_right(colorize_levels(toupper(level_name)), 5)} [{timestamp}] {msg}"
)
```

**Method** `format_event()`:

*Usage:*

```
LayoutGlue$format_event(event)
```

**Method** `set_fmt()`:

*Usage:*

```
LayoutGlue$set_fmt(x)
```

**Method** `set_colors()`:

*Usage:*

```
LayoutGlue$set_colors(x)
```

**Method** `toString()`:

*Usage:*

```
LayoutGlue$toString()
```

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
LayoutGlue$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

`lgr` exports a number of formatting utility functions that are useful for layout glue: [colorize\\_levels\(\)](#), [pad\\_left\(\)](#), [pad\\_right\(\)](#).

Other Layouts: [LayoutFormat](#), [LayoutJson](#), [Layout](#)

## Examples

```
lg <- get_logger("test")$
  set_appenders(AppenderConsole$new())$
  set_propagate(FALSE)
```

```
lg$appenders[[1]]$set_layout(LayoutGlue$new())
lg$fatal("test")
```

```
# All fields of the LogEvent are available, even custom ones
```

```
lg$appenders[[1]]$layout$set_fmt(
  "{logger} {level_name}({level}) {caller}: {toupper(msg)} {{custom: {custom}}}"
)
lg$fatal("test", custom = "foobar")
lg$config(NULL) # reset logger config
```



---

LayoutJson

*Format LogEvents as JSON*

---

## Description

A format for formatting LogEvents as `jsonlines` log files. This provides a nice balance between human- and machine-readability.

## Super class

`lgr::Layout` -> LayoutJson

## Active bindings

`toJSON_args` a list of values passed on to `jsonlite::toJSON()`

`timestamp_fmt` Used by `$format_event()` to format timestamps.

## Methods

### Public methods:

- `LayoutJson$new()`
- `LayoutJson$format_event()`
- `LayoutJson$set_toJSON_args()`
- `LayoutJson$set_timestamp_fmt()`
- `LayoutJson$toString()`
- `LayoutJson$parse()`
- `LayoutJson$read()`
- `LayoutJson$clone()`

### Method `new()`:

*Usage:*

```
LayoutJson$new(toJSON_args = list(auto_unbox = TRUE), timestamp_fmt = NULL)
```

### Method `format_event()`:

*Usage:*

```
LayoutJson$format_event(event)
```

### Method `set_toJSON_args()`: Set arguments to pass on to `jsonlite::toJSON()`

*Usage:*

```
LayoutJson$set_toJSON_args(x)
```

*Arguments:*

`x` a named list

### Method `set_timestamp_fmt()`: Set a format that this Layout will apply to timestamps.

*Usage:*

LayoutJson\$set\_timestamp\_fmt(x)

*Arguments:*

- x • NULL (the default): formatting of the timestamp is left to `jsonlite::toJSON()`,
  - a character scalar as for `format.POSIXct()`, or
  - a function that returns a vector of the same length as its (`POSIXct`) input. The returned vector can be of any type supported by `jsonlite::toJSON()`, but should usually be character.

**Method** toString():

*Usage:*

LayoutJson\$toString()

**Method** parse():

*Usage:*

LayoutJson\$parse(file)

**Method** read():

*Usage:*

LayoutJson\$read(file, threshold = NA\_integer\_, n = 20L)

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

LayoutJson\$clone(deep = FALSE)

*Arguments:*

deep Whether to make a deep clone.

## See Also

`read_json_lines()`, <https://jsonlines.org/>

Other Layouts: [LayoutFormat](#), [LayoutGlue](#), [Layout](#)

## Examples

```
# setup a dummy LogEvent
event <- LogEvent$new(
  logger = Logger$new("dummy logger"),
  level = 200,
  timestamp = Sys.time(),
  caller = NA_character_,
  msg = "a test message",
  custom_field = "LayoutJson can handle arbitrary fields"
)

# Default settings show all event fals
lo <- LayoutJson$new()
lo$format_event(event)
```

---

 LogEvent

*LogEvents - The atomic unit of logging*


---

### Description

A LogEvent is a single unit of data that should be logged. LogEvents are usually created by a [Logger](#), and then processed by one more [Appenders](#). They do not need to be instantiated manually except for testing and experimentation; however, if you plan on writing your own Appenders or Layouts you need to understand LogEvents.

### Public fields

level integer. The [log\\_level](#) / priority of the LogEvent. Use the active binding level\_name to get the character representation instead.

timestamp [POSIXct](#). The time when then the LogEvent was created.

caller character. The name of the calling function.

msg character. The log message.

.logger [Logger](#). A reference to the Logger that created the event (equivalent to `get_logger(event$logger)`).

### Active bindings

values list. All values stored in the LogEvent, including all *custom fields*, but not including `event$.logger`.

level\_name character. The [log\\_level](#) / priority of the LogEvent labelled according to `getOption("lgr.log_levels")`

logger character scalar. The name of the Logger that created this event, equivalent to `event$.logger$name`

### Methods

#### Public methods:

- [LogEvent\\$new\(\)](#)
- [LogEvent\\$clone\(\)](#)

**Method new():** The arguments to `LogEvent$new()` directly translate to the fields stored in the LogEvent. Usually these values will be scalars, but (except for "logger") they can also be vectors if they are all of the same length (or scalars that will be recycled). In this case the event will be treated by the [Appenders](#) and [Layouts](#) as if several separate events.

*Usage:*

```
LogEvent$new(
  logger,
  level = 400,
  timestamp = Sys.time(),
  caller = NA,
  msg = NA,
  ...
)
```

*Arguments:*

logger, level, timestamp, caller, msg see **Public fields**.

... All named arguments in ... will be added to the LogEvent as **custom fields**. You can store arbitrary R objects in LogEvents this way, but not all Appenders will support them. See [AppenderJson](#) for

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

```
LogEvent$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

**See Also**

[as.data.frame.LogEvent\(\)](#)

Other docs relevant for extending lgr: [as\\_LogEvent\(\)](#), [event\\_list\(\)](#), [standardize\\_threshold\(\)](#)

**Examples**

```
lg <- get_logger("test")
lg$error("foo bar")

# The last LogEvent produced by a Logger is stored in its `last_event` field
lg$last_event # formatted console output
lg$last_event$values # values stored in the event

# Also contains the Logger that created it as .logger
lg$last_event$logger
# equivalent to
lg$last_event$.logger$name

# This is really a reference to the complete Logger, so the following is
# possible (though nonsensical)
lg$last_event$.logger$last_event$msg
identical(lg, lg$last_event$.logger)
lg$config(NULL) # reset logger config
```

---

Logger

*Loggers*

---

**Description**

A Logger produces a [LogEvent](#) that contains a log message along with metadata (timestamp, calling function, ...) and dispatches it to one or more [Appenders](#) which are responsible for the output (console, file, ...) of the event. **lgr** comes with a single pre-configured Logger called the root Logger that can be accessed via `lgr$<. . .>`. Instantiation of new Loggers is done with [get\\_logger\(\)](#). It is advisable to instantiate a separate Logger with a descriptive name for each package/script in which you use **lgr**.

**Super class**

`lgr::Filterable` -> Logger

**Active bindings**

`name` A character scalar. The unique name of each logger, which also includes the names of its ancestors (separated by /).

`threshold` integer scalar. The threshold of the Logger, or if it NULL the threshold it inherits from its closest ancestor with a non-NULL threshold

`propagate` A TRUE or FALSE. The unique name of each logger, which also includes the names of its ancestors (separated by /).

`ancestry` A named logical vector of containing the propagate value of each Logger upper the inheritance tree. The names are the names of the appenders. `ancestry` is an S3 class with a custom `format()/print()` method, so if you want to use the plain logical vector use `unclass(lg$ancestry)`

`parent` a Logger. The direct ancestor of the Logger.

`last_event` The last LogEvent produced by the current Logger

`appenders` a list of all [Appenders](#) of the Logger

`inherited_appenders` A list of all appenders that the Logger inherits from its ancestors

`exception_handler` a function. See `$set_exception_handler` and `$handle_exception`

**Methods****Public methods:**

- `Logger$new()`
- `Logger$log()`
- `Logger$fatal()`
- `Logger$error()`
- `Logger$warn()`
- `Logger$info()`
- `Logger$debug()`
- `Logger$trace()`
- `Logger$list_log()`
- `Logger$config()`
- `Logger$add_appender()`
- `Logger$remove_appender()`
- `Logger$handle_exception()`
- `Logger$set_exception_handler()`
- `Logger$set_propagate()`
- `Logger$set_threshold()`
- `Logger$set_appenders()`
- `Logger$spawn()`

**Method new():** **Loggers should never be instantiated directly with `Logger$new()`** but rather via `get_logger("name")`. This way new Loggers are registered in a global namespace which ensures uniqueness and facilitates inheritance between Loggers. If "name" does not exist, a new Logger with that name will be created, otherwise the function returns a Reference to the existing Logger.

name is potentially a "/" separated hierarchical value like foo/bar/baz. Loggers further down the hierarchy are descendants of the loggers above and (by default) inherit threshold and Appenders from their ancestors.

*Usage:*

```
Logger$new(
  name = "(unnamed logger)",
  appenders = list(),
  threshold = NULL,
  filters = list(),
  exception_handler = default_exception_handler,
  propagate = TRUE
)
```

*Arguments:*

name, appenders, threshold, filters, exception\_handler, propagate See section Active bindings.

**Method log():** Log an event.

If level passes the Logger's threshold a new `LogEvent` with level, msg, timestamp and caller is created. If the new `LogEvent` also passes the Loggers `Filters`, it is be dispatched to the relevant `Appenders`.

*Usage:*

```
Logger$log(level, msg, ..., timestamp = Sys.time(), caller = get_caller(-7))
```

*Arguments:*

level a character or integer scalar. See `log_levels`.

msg character. A log message. If unnamed arguments are supplied in ..., msg is passed on to `base::sprintf()` (which means "%" have to be escaped), otherwise msg is left as-is.

... *unnamed* arguments in ... must be character scalars and are passed to `base::sprintf()`.

*Named* arguments must have unique names but can be arbitrary R objects that are passed to `LogEvent$new()` and will be turned into custom fields.

timestamp `POSIXct`. Timestamp of the event.

caller a character scalar. The name of the calling function.

**Method fatal():** Log an Event fatal priority

*Usage:*

```
Logger$fatal(msg, ..., caller = get_caller(-8L))
```

*Arguments:*

msg, ..., caller see `$log()`

**Method error():** Log an Event error priority

*Usage:*

```
Logger$error(msg, ..., caller = get_caller(-8L))
```

*Arguments:*

msg, ..., caller see \$log()

**Method warn():** Log an Event warn priority

*Usage:*

```
Logger$warn(msg, ..., caller = get_caller(-8L))
```

*Arguments:*

msg, ..., caller see \$log()

**Method info():** Log an Event info priority

*Usage:*

```
Logger$info(msg, ..., caller = get_caller(-8L))
```

*Arguments:*

msg, ..., caller see \$log()

**Method debug():** Log an Event debug priority

*Usage:*

```
Logger$debug(msg, ..., caller = get_caller(-8L))
```

*Arguments:*

msg, ..., caller see \$log()

**Method trace():** Log an Event trace priority

*Usage:*

```
Logger$trace(msg, ..., caller = get_caller(-8L))
```

*Arguments:*

msg, ..., caller see \$log()

**Method list\_log():** `list_log()` is a shortcut for `do.call(Logger$log, x)`. See <https://github.com/s-fleck/joblog> for an R package that leverages this feature to create custom log event types for tracking the status of cron jobs.

*Usage:*

```
Logger$list_log(x)
```

*Arguments:*

x a named list that must at least contain the named elements level and timestamp

*Examples:*

```
lg <- get_logger("test")
lg$list_log(list(level = 400, msg = "example"))
```

**Method config():** Load a Logger configuration.

*Usage:*

```
Logger$config(cfg, file, text, list)
```

*Arguments:*

`cfg` • a special list object with any or all of the the following elements: `appenders`, `threshold`, `filters`, `propagate`, `exception_handler`,

- the path to a YAML/JSON config file,
- a character scalar containing YAML/JSON,
- NULL (to reset the logger config to the default/unconfigured state)

`file`, `text`, `list` can be used as an alternative to `cfg` that enforces that the supplied argument is of the specified type. See [logger\\_config](#) for details.

**Method** `add_appender()`: Add an Appender to the Logger

*Usage:*

```
Logger$add_appender(appender, name = NULL)
```

*Arguments:*

`appender` a single [Appender](#)

`name` a character scalar. Optional but recommended.

*Examples:*

```
lg <- get_logger("test")
lg$add_appender(AppenderConsole$new(), name = "myconsole")
lg$appenders[[1]]
lg$appenders$myconsole
lg$remove_appender("myconsole")
lg$config(NULL) # reset config
```

**Method** `remove_appender()`: remove an appender

*Usage:*

```
Logger$remove_appender(pos)
```

*Arguments:*

`pos` integer index or character name of the Appender(s) to remove

**Method** `handle_exception()`: To prevent errors in the logging logic from crashing the whole script, Loggers pass errors they encounter to an exception handler. The default behaviour is to demote errors to [warnings](#). See also `set_exception_handler()`.

*Usage:*

```
Logger$handle_exception(expr)
```

*Arguments:*

`expr` expression to be evaluated.

**Method** `set_exception_handler()`: Set the exception handler of a logger

*Usage:*

```
Logger$set_exception_handler(fun)
```

*Arguments:*

`fun` a function with the single argument `e` (an error [condition](#))

*Examples:*



```
lgr$info(stop("this produces a warning instead of an error"))
```

**Method** `set_propagate()`: Should a Logger propagate events to the Appenders of its ancestors?

*Usage:*

```
Logger$set_propagate(x)
```

*Arguments:*

x TRUE or FALSE. Should [LogEvents](#) be passed on to the appenders of the ancestral Loggers?

**Method** `set_threshold()`: Set the minimum log level of events that a Logger should process

*Usage:*

```
Logger$set_threshold(level)
```

*Arguments:*

level character or integer scalar. The minimum [log level](#) that triggers this Logger

**Method** `set_appenders()`: Set the Logger's Appenders

*Usage:*

```
Logger$set_appenders(x)
```

*Arguments:*

x single [Appender](#) or a list thereof. Appenders control the output of a Logger. Be aware that a Logger also inherits the Appenders of its ancestors (see `vignette("lgr", package = "lgr")` for more info about Logger inheritance).

**Method** `spawn()`: Spawn a child Logger. This is very similar to using `get_logger()`, but can be useful in some cases where Loggers are created programmatically

*Usage:*

```
Logger$spawn(name)
```

*Arguments:*

name character vector. Name of the child logger `get_logger("foo/bar")$spawn("baz")` is equivalent to `get_logger("foo/bar/baz")`

## Note

If you are a package developer you should define a new Logger for each package, but you do not need to configure it. The user of the package should decide how and where to output logging, usually by configuring the root Logger (new Appenders added/removed, Layouts modified, etc...).

## See Also

[glue](#)

[get\\_logger\(\)](#)

**Examples**

```

# lgr::lgr is the root logger that is always available
lgr$info("Today is a good day")
lgr$fatal("This is a serious error")

# Loggers use sprintf() for string formatting by default
lgr$info("Today is %s", Sys.Date() )

# If no unnamed `...` are present, msg is not passed through sprintf()
lgr$fatal("100% bad") # so this works
lgr$fatal("%s%% bad", 100) # if you use unnamed arguments, you must escape %

# You can create new loggers with get_logger()
tf <- tempfile()
lg <- get_logger("mylogger")$set_appenders(AppenderFile$new(tf))

# The new logger passes the log message on to the appenders of its parent
# logger, which is by default the root logger. This is why the following
# writes not only the file 'tf', but also to the console.
lg$fatal("blubb")
readLines(tf)

# This logger's print() method depicts this relationship.
child <- get_logger("lg/child")
print(child)
print(child$name)

# use formatting strings and custom fields
tf2 <- tempfile()
lg$add_appender(AppenderFile$new(tf2, layout = LayoutJson$new()))
lg$info("Not all %s support custom fields", "appenders", type = "test")
cat(readLines(tf), sep = "\n")
cat(readLines(tf2), sep = "\n")

# cleanup
unlink(c(tf, tf2))
lg$config(NULL) # reset logger config

# LoggerGlue
# You can also create a new logger that uses the awesome glue library for
# string formatting instead of sprintf

if (requireNamespace("glue")){

  lg <- get_logger_glue("glue")
  lg$fatal("blah ", "fizz is set to: {fizz}", foo = "bar", fizz = "buzz")
  # prevent creation of custom fields with prefixing a dot
  lg$fatal("blah ", "fizz is set to: {.fizz}", foo = "bar", .fizz = "buzz")

  #' # completely reset 'glue' to an unconfigured vanilla Logger
  get_logger("glue", reset = TRUE)
}

```

```

}

# Configuring a Logger
lg <- get_logger("test")
lg$config(NULL) # resets logger to unconfigured state

# With setters
lg$
  set_threshold("error")$
  set_propagate(FALSE)$
  set_appenders(AppenderConsole$new(threshold = "info"))

lg$config(NULL)

# With a list
lg$config(list(
  threshold = "error",
  propagate = FALSE,
  appenders = list(AppenderConsole$new(threshold = "info"))
))

lg$config(NULL) # resets logger to unconfigured state

# Via YAML
cfg <- "
Logger:
  threshold: error
  propagate: false
  appenders:
    AppenderConsole:
      threshold: info
"

lg$config(cfg)
lg$config(NULL)

## -----
## Method `Logger$list_log`
## -----

lg <- get_logger("test")
lg$list_log(list(level = 400, msg = "example"))

## -----
## Method `Logger$add_appender`
## -----

lg <- get_logger("test")
lg$add_appender(AppenderConsole$new(), name = "myconsole")
lg$appenders[[1]]
lg$appenders$myconsole
lg$remove_appender("myconsole")

```

```
lg$config(NULL) # reset config

## -----
## Method `Logger$set_exception_handler`
## -----

lgr$info(stop("this produces a warning instead of an error"))
```

---

 LoggerGlue

*LoggerGlue*


---

## Description

LoggerGlue uses `glue::glue()` instead of `base::sprintf()` to construct log messages. `glue` is a very well designed package for string interpolation. It makes composing log messages more flexible and comfortable at the price of an additional dependency and slightly less performance than `sprintf()`.

## Details

`glue()` lets you define temporary named variables inside the call. As with the normal `Logger`, these named arguments get turned into custom fields; however, you can suppress this behaviour by making named argument start with a `."`. Please refer to `vignette("lgr", package = "lgr")` for examples.

## Super classes

```
lgr::Filterable -> lgr::Logger -> LoggerGlue
```

## Methods

### Public methods:

- `LoggerGlue$fatal()`
- `LoggerGlue$error()`
- `LoggerGlue$warn()`
- `LoggerGlue$info()`
- `LoggerGlue$debug()`
- `LoggerGlue$trace()`
- `LoggerGlue$log()`
- `LoggerGlue$list_log()`
- `LoggerGlue$spawn()`

### Method `fatal()`:

*Usage:*

```
LoggerGlue$fatal(..., caller = get_caller(-8L), .envir = parent.frame())
```

**Method error():***Usage:*

```
LoggerGlue$error(..., caller = get_caller(-8L), .envir = parent.frame())
```

**Method warn():***Usage:*

```
LoggerGlue$warn(..., caller = get_caller(-8L), .envir = parent.frame())
```

**Method info():***Usage:*

```
LoggerGlue$info(..., caller = get_caller(-8L), .envir = parent.frame())
```

**Method debug():***Usage:*

```
LoggerGlue$debug(..., caller = get_caller(-8L), .envir = parent.frame())
```

**Method trace():***Usage:*

```
LoggerGlue$trace(..., caller = get_caller(-8L), .envir = parent.frame())
```

**Method log():***Usage:*

```
LoggerGlue$log(  
  level,  
  ...,  
  timestamp = Sys.time(),  
  caller = get_caller(-7),  
  .envir = parent.frame()  
)
```

**Method list\_log():***Usage:*

```
LoggerGlue$list_log(x)
```

**Method spawn():***Usage:*

```
LoggerGlue$spawn(name)
```

---

 logger\_config      *Logger Configuration Objects*


---

**Description**

logger\_config() is an S3 constructor for logger\_config objects that can be passed to the \$config method of a [Logger](#). You can just pass a normal list instead, but using this constructor is a more formal way that includes additional argument checking.

**Usage**

```
logger_config(
  appenders = NULL,
  threshold = NULL,
  filters = NULL,
  exception_handler = NULL,
  propagate = TRUE
)

as_logger_config(x)

## S3 method for class 'list'
as_logger_config(x)

## S3 method for class 'character'
as_logger_config(x)
```

**Arguments**

appenders	see <a href="#">Logger</a>
threshold	see <a href="#">Logger</a>
filters	see <a href="#">Logger</a>
exception_handler	see <a href="#">Logger</a>
propagate	see <a href="#">Logger</a>
x	any R object. Especially: <ul style="list-style-type: none"> <li>• A character scalar. This can either be the path to a YAML file or a character scalar containing valid YAML</li> <li>• a list containing the elements appenders, threshold, exception_handler, propagate and filters. See the section <i>Fields</i> in <a href="#">Logger</a> for details.</li> <li>• a Logger object, to clone its configuration.</li> </ul>

**Value**

a list with the subclass "logger\_config"  
 a logger\_config object

**See Also**

<https://yaml.org/>

---

logger_index	<i>Return a data.frame of all registered loggers</i>
--------------	--

---

**Description**

Return a data.frame of all registered loggers

**Usage**

```
logger_index()
```

**Value**

a logger\_index data.frame

**See Also**

[logger\\_tree\(\)](#) for a more visual representation of registered loggers

**Examples**

```
get_logger("tree/leaf")
get_logger("shrub/leaf")
get_logger("plant/shrub/leaf")
logger_index()
```

---

logger_tree	<i>Logger Tree</i>
-------------	--------------------

---

**Description**

Displays a tree structure of all registered Loggers.

**Usage**

```
logger_tree()
```

**Value**

data.frame with subclass "logger\_tree"

**Symbology**

- unconfigured Loggers are displayed in gray (if your terminal supports colors and you have the package **crayon** installed).
- If a logger's threshold is set, it is displayed in square brackets next to its name (reminder: if the threshold is not set, it is inherited from next logger up the logger tree).
- If a logger's propagate field is set to FALSE an red hash (#) sign is displayed in front of the logger name, to imply that it does not pass LogEvents up the tree.

**See Also**

[logger\\_index\(\)](#) for a tidy data . frame representation of all registered loggers

**Examples**

```
get_logger("fancymodel")
get_logger("fancymodel/shiny")$
  set_propagate(FALSE)

get_logger("fancymodel/shiny/ui")$
  set_appenders(AppenderConsole$new())

get_logger("fancymodel/shiny/server")$
  set_appenders(list(AppenderConsole$new(), AppenderConsole$new()))$
  set_threshold("trace")

get_logger("fancymodel/plumber")

if (requireNamespace("cli")){
  print(logger_tree())
}
```

---

pad\_right

*Pad Character Vectors*

---

**Description**

Pad Character Vectors

**Arguments**

x	a character vector
width	integer scalar. target string width
pad	character scalar. the symbol to pad with



**Examples**

```
pad_left("foo", 5)
pad_right("foo", 5, ".")
pad_left(c("foo", "foooooo"), pad = ".")
```

---

print.Appender	<i>Print an Appender object</i>
----------------	---------------------------------

---

**Description**

The `print()` method for Loggers displays the most important aspects of the Appender.

**Usage**

```
## S3 method for class 'Appender'
print(x, color = requireNamespace("crayon", quietly = TRUE), ...)
```

**Arguments**

<code>x</code>	any R Object
<code>color</code>	TRUE or FALSE: Output with color? Requires the Package <b>crayon</b>
<code>...</code>	ignored

**Value**

`print()` returns `x` (invisibly), `format()` returns a character vector.

**Examples**

```
# print most important details of logger
print(lgr$console)
```

---

print.LogEvent	<i>Print or Format Logging Data</i>
----------------	-------------------------------------

---

**Description**

Print or Format Logging Data

**Usage**

```

## S3 method for class 'LogEvent'
print(
  x,
  fmt = "%L [%t] %m %f",
  timestamp_fmt = "%Y-%m-%d %H:%M:%S",
  colors = getOption("lgr.colors"),
  log_levels = getOption("lgr.log_levels"),
  pad_levels = "right",
  ...
)

## S3 method for class 'LogEvent'
format(
  x,
  fmt = "%L [%t] %m %f",
  timestamp_fmt = "%Y-%m-%d %H:%M:%S",
  colors = NULL,
  log_levels = getOption("lgr.log_levels"),
  pad_levels = "right",
  ...
)

```

**Arguments**

x	a <a href="#">LogEvent</a>
fmt	A character scalar that may contain any of the tokens listed bellow in the section Format Tokens.
timestamp_fmt	see <a href="#">format.POSIXct()</a>
colors	A list of functions that will be used to color the log levels (likely from <a href="#">crayon::crayon</a> ).
log_levels	named integer vector of valid log levels
pad_levels	right, left or NULL. Whether or not to pad the log level names to the same width on the left or right side, or not at all.
...	ignored

**Value**

x for print() and a character scalar for format()

**Format Tokens**

- %t The timestamp of the message, formatted according to timestamp\_fmt)
- %l the log level, lowercase character representation
- %L the log level, uppercase character representation
- %k the log level, first letter of lowercase character representation

%K the log level, first letter of uppercase character representation  
 %n the log level, integer representation  
 %g the name of the logger  
 %p the PID (process ID). Useful when logging code that uses multiple threads.  
 %c the calling function  
 %m the log message  
 %f all custom fields of x in a pseudo-JSON like format that is optimized for human readability and console output  
 %j all custom fields of x in proper JSON. This requires that you have **jsonlite** installed and does not support colors as opposed to %f

### Examples

```

# standard fields can be printed using special tokens
x <- LogEvent$new(
  level = 300, msg = "a test event", caller = "testfun()", logger = lgr
)
print(x)
print(x, fmt = c("%t (%p) %c: %n - %m"))
print(x, colors = NULL)

# custom values
y <- LogEvent$new(
  level = 300, msg = "a gps track", logger = lgr,
  waypoints = 10, location = "Austria"
)

# default output with %f
print(y)

# proper JSON output with %j
if (requireNamespace("jsonlite")){
  print(y, fmt = "%L [%t] %m %j")
}

```

---

print.Logger

*Print a Logger Object*

---

### Description

The `print()` method for Loggers displays the most important aspects of the Logger.

You can also print just the ancestry of a Logger which can be accessed with `logger$ancestry()`. This returns a named character vector whose names correspond to the names of the Loggers logger inherits from. The TRUE/FALSE status of its elements correspond to the propagate values of these Loggers.

**Usage**

```
## S3 method for class 'Logger'
print(x, color = requireNamespace("crayon", quietly = TRUE), ...)

## S3 method for class 'Logger'
format(x, color = FALSE, ...)

## S3 method for class 'ancestry'
print(x, color = requireNamespace("crayon", quietly = TRUE), ...)

## S3 method for class 'ancestry'
format(x, color = FALSE, ...)
```

**Arguments**

x	any R Object
color	TRUE or FALSE: Output with color? Requires the Package <b>crayon</b>
...	ignored

**Value**

print() returns x (invisibly), format() returns a character vector.

**Examples**

```
# print most important details of logger
print(lgr)
# print only the ancestry of a logger
lg <- get_logger("AegonV/Aerys/Rheagar/Aegon")
get_logger("AegonV/Aerys/Rheagar")$set_propagate(FALSE)

print(lg$ancestry)
unclass(lg$ancestry)
```

---

print.logger\_tree      *Print Logger Trees*

---

**Description**

Print Logger Trees

**Usage**

```
## S3 method for class 'logger_tree'
print(x, color = requireNamespace("crayon", quietly = TRUE), ...)

## S3 method for class 'logger_tree'
format(x, color = FALSE, ...)
```

**Arguments**

x a [logger\\_tree](#)

color logical scalar. If TRUE terminal output is colored via the package **crayon**?

... passed on to [cli::tree\(\)](#)

**Value**

x (invisibly)

---

read_json_lines	<i>Read a JSON logfile</i>
-----------------	----------------------------

---

**Description**

Read a JSON logfile

**Usage**

```
read_json_lines(file, ...)
```

**Arguments**

file character scalar. path to a JSON logfile (one JSON object per line)

... passed on to [jsonlite::stream\\_in\(\)](#)

**Value**

a data.frame

**See Also**

[LayoutJson](#)

## Description

`lgr` provides convenience functions managing the root `Logger`. These are designed chiefly for interactive use and are less verbose than their R6 method counterparts.

`threshold()` sets or retrieves the threshold for an [Appender](#) or [Logger](#) (the minimum level of log messages it processes). It's `target` defaults to the root logger. (equivalent to `lgr::lgr$threshold` and `lgr::lgr$set_threshold`)

`console_threshold()` is a shortcut to set the threshold of the root loggers [AppenderConsole](#), which is usually the only `Appender` that manages console output for a given R session. (equivalent to `lgr::lgr$appenders$console$threshold` and `lgr::lgr$appenders$console$set_threshold`)

`add_appender()` and `remove_appender()` add `Appenders` to `Loggers` and other `Appenders`. (equivalent to `lgr::lgr$add_appender` and `lgr::lgr$remove_appender`)

`show_log()` displays the last `n` log entries of an `Appender` (or a `Logger` with such an `Appender` attached) with a `$show()` method. Most, but not all `Appenders` support this function (try [AppenderFile](#) or [AppenderBuffer](#)).

`show_data()` and `show_dt()` work similar to `show_log()`, except that they return the log as `data.frame` or `data.table` respectively. Only `Appenders` that log to formats that can easily be converted to `data.frames` are supported (try [AppenderJson](#) or [AppenderBuffer](#)).

The easiest way to try out this features is by adding an `AppenderBuffer` to the root logger with `basic_config(memory = TRUE)`.

## Usage

```
log_exception(code, logfun = lgr$fatal, caller = get_caller(-3))
```

```
threshold(level, target = lgr::lgr)
```

```
console_threshold(level, target = lgr::lgr$appenders$console)
```

```
add_appender(appender, name = NULL, target = lgr::lgr)
```

```
remove_appender(pos, target = lgr::lgr)
```

```
show_log(threshold = NA_integer_, n = 20L, target = lgr::lgr)
```

```
show_dt(target = lgr::lgr)
```

```
show_data(target = lgr::lgr)
```

## Arguments

```
code          Any R code
```

logfun	a function for processing the log request, usually <code>lgr\$info()</code> , <code>lgr\$debug()</code> , etc... .
caller	a character scalar. The name of the calling function
level	integer or character scalar: the desired log level
target	a <a href="#">Logger</a> or <a href="#">Appender</a> or the name of a Logger as character scalar
appender	an Appender
name	character scalar. An optional name for the new Appender.
pos	integer index or character names of the appenders to remove
threshold	character or integer scalar. The minimum <a href="#">log level</a> that should be processed by the root logger.
n	integer scalar. Show only the last n log entries that match threshold

### Value

`threshold()` and `console_threshold()` return the [log\\_level](#) of target as integer (invisibly)

`add_appender()` and `remove_appender()` return target.

`show_log()` prints to the console and returns whatever the target Appender's `$show()` method returns, usually a character vector, `data.frame` or `data.table` (invisibly).

`show_data()` always returns a `data.frame` and `show_dt()` always returns a `data.table`.

### Examples

```
# Get and set the threshold of the root logger
threshold("error")
threshold()
lgr$info("this will be suppressed")
lgr$error("an important error message")

# you can also specify a target to modify other loggers
lg <- get_logger("test")
threshold("fatal", target = lg)
threshold(target = lg)

# If a Logger's threshold is not set, the threshold is inherited from
# its parent, in this case the root logger (that we set to error/200 before)
threshold(NULL, target = lg)
threshold(target = lg)

# Alternative R6 API for getting/setting thresholds
lg$set_threshold("info")
lg$threshold
lg$set_threshold(300)
lg$threshold
lg$set_threshold(NULL)
lg$threshold

# cleanup
lgr$config(NULL)
```

```

lgr$config(NULL)

# add Appenders to a Logger
add_appender(AppenderConsole$new(), "second_console_appender")
lgr$fatal("Multiple console appenders are a bad idea")
remove_appender("second_console_appender")
lgr$info("Good that we defined an appender name, so it's easy to remove")

# Reconfigure the root logger
basic_config(memory = TRUE)

# log some messages
lgr$info("a log message")
lgr$info("another message with data", data = 1:3)

show_log()
show_data()

# cleanup
lgr$config(NULL)

```

---

standardize\_threshold *Standardize User-Input Log Levels to Their Integer Representation*

---

## Description

These are helper functions for verifying log levels and converting them from their character to their integer representations. This is primarily useful if you want to build your own [Loggers](#), [Appenders](#) or [Layouts](#) and need to handle log levels in a way that is consistent with **lgr** .

## Usage

```

standardize_threshold(
  x,
  log_levels = c(getOption("lgr.log_levels"), c(all = NA_integer_, off = 0L))
)

is_threshold(x)

standardize_log_level(x, log_levels = getOption("lgr.log_levels"))

is_log_level(x)

standardize_log_levels(x, log_levels = getOption("lgr.log_levels"))

is_log_levels(x)

```



**Arguments**

`x` a character or integer scalar, or vector for `standardize_log_levels`  
`log_levels` named integer vector of valid log levels

**Value**

An unnamed integer vector

**See Also**

Other docs relevant for extending lgr: [LogEvent](#), [as\\_LogEvent\(\)](#), [event\\_list\(\)](#)

**Examples**

```
standardize_threshold("info")
standardize_threshold("all")
is_threshold("all")
is_threshold("foobar")

standardize_log_level("info")
# all is a valid threshold, but not a valid log level
try(is.na(standardize_log_level("all")))
is_log_level("all")

# standardized_log_level intentionally only works with scalars, because many
# functions require scalar log level inputs
try(standardize_log_level(c("info", "fatal")))

# You can still use standardize_log_levels() (plural) to work with vectors
standardize_log_levels(c("info", "fatal"))
```

---

string\_repr

*Short string representation for R objects*


---

**Description**

This is inspired by the python function `repr` and produces a short string representation of any R object that is suitable for logging and error messages. It is a generic so you can implement methods for custom S3 objects.

**Usage**

```
string_repr(x, width = 32, ...)

## S3 method for class ``function``
string_repr(x, width = 32L, ...)
```

```
## S3 method for class 'data.frame'  
string_repr(x, width = 32L, ...)  
  
## S3 method for class 'matrix'  
string_repr(x, width = 32L, ...)  
  
## Default S3 method:  
string_repr(x, width = 32L, ...)
```

### Arguments

x	Any R object.
width	a scalar integer
...	passed on to methods

### Value

a scalar character

### Examples

```
string_repr(iris)  
string_repr(LETTERS)  
string_repr(LETTERS, 10)
```

---

suspend_logging	<i>Suspend All Logging</i>
-----------------	----------------------------

---

### Description

Completely disable logging for all loggers. This is for example useful for automated test code. `suspend_logging()` globally disables all logging with `lgr` until `unsuspend_logging()` is invoked, while `without_logging()` and `with_logging()` temporarily disable/enable logging.

### Usage

```
suspend_logging()  
  
unsuspend_logging()  
  
without_logging(code)  
  
with_logging(code)
```

### Arguments

code	Any R code
------	------------

**Value**

suspend\_logging() and unsuspend\_logging() return NULL (invisibly), without\_logging() and with\_logging() returns whatever code returns.

**Examples**

```
lg <- get_logger("test")

# temporarily disable logging
lg$fatal("foo")
without_logging({
  lg$info("everything in this codeblock will be suppressed")
  lg$fatal("bar")
})

# globally disable logging
suspend_logging()
lg$fatal("bar")
with_logging(lg$fatal("foo")) # log anyways

# globally enable logging again
unsuspend_logging()
lg$fatal("foo")
```

---

toString.LogEvent	<i>Convert a LogEvent to a character string</i>
-------------------	---

---

**Description**

Convert a LogEvent to a character string

**Usage**

```
## S3 method for class 'LogEvent'
toString(x, ...)
```

**Arguments**

x	a <a href="#">LogEvent</a>
...	ignored

**Value**

a character scalar

**Examples**

```
toString(LogEvent$new(logger = lgr::lgr))
```

---

 use\_logger

*Setup a Simple Logger for a Package*


---

### Description

This gives you a minimal logger with no appenders that you can use inside your package under the name `lg` (e.g. `lg$fatal("test")`). `use_logger()` does not modify any files but only prints code for you to copy and paste.

### Usage

```
use_logger(
  pkg = desc::desc_get("Package", rprojroot::find_package_root_file("DESCRIPTION"))[[1]]
)
```

### Arguments

`pkg` character scalar. Name of the package. The default is to try to get the Package name automatically using the packages **rprojroot** and **desc**

### Value

a character scalar containing R code.

### Examples

```
use_logger("testpkg")
```

---

 with\_log\_level

*Inject Values into Logging Calls*


---

### Description

`with_log_level` temporarily overrides the log level of all [LogEvents](#) created by target [Logger](#).

### Usage

```
with_log_level(level, code, logger = lgr::lgr)

with_log_value(values, code, logger = lgr::lgr)
```

**Arguments**

level	integer or character scalar: the desired log level
code	Any R code
logger	a <a href="#">Logger</a> or the name of one (see <a href="#">get_logger()</a> ). Defaults to the root logger (lgr::lgr).
values	a named list of values to be injected into the logging calls

**Details**

These functions abuses lgr's filter mechanic to modify LogEvents in-place before they passed on the Appenders. Use with care as they can produce hard to reason about code.

**Value**

whatever code would return

**Examples**

```
with_log_level("warn", {
  lgr$info("More important than it seems")
  lgr$fatal("Really not so bad")
})
with_log_value(
  list(msg = "overriden msg"), {
    lgr$info("bar")
    lgr$fatal("FOO")
  })
```

# Index

- \* **Appenders**
  - AppenderBuffer, 3
  - AppenderConsole, 5
  - AppenderFile, 6
  - AppenderFileRotating, 9
  - AppenderFileRotatingDate, 11
  - AppenderFileRotatingTime, 13
  - AppenderTable, 17
- \* **Layouts**
  - Layout, 35
  - LayoutFormat, 36
  - LayoutGlue, 39
  - LayoutJson, 41
- \* **abstract classes**
  - AppenderMemory, 15
  - AppenderTable, 17
  - Filterable, 27
- \* **developer tools**
  - CannotInitializeAbstractClassError, 22
- \* **docs relevant for extending lgr**
  - as\_LogEvent, 20
  - event\_list, 26
  - LogEvent, 43
  - standardize\_threshold, 64
- \* **formatting utils**
  - colorize\_levels, 23
  - label\_levels, 34
  - .obj (EventFilter), 24
- add\_appender (simple\_logging), 62
- add\_log\_levels (get\_log\_levels), 33
- Appender, 4–6, 8, 11, 12, 14, 17, 18, 21, 24, 28, 36, 48, 49, 62, 63
- AppenderBuffer, 3, 4, 6, 8, 11, 12, 14, 15, 18, 21, 26, 62
- AppenderConsole, 5, 5, 8, 11, 12, 14, 18, 62
- AppenderFile, 5, 6, 6, 9, 11, 12, 14, 18, 21, 35, 62
- AppenderFileRotating, 5, 6, 8, 9, 12, 14, 18
- AppenderFileRotatingDate, 5, 6, 8, 11, 11, 14, 18
- AppenderFileRotatingTime, 5, 6, 8, 11, 12, 13, 18
- AppenderJson, 6, 21, 44, 62
- AppenderJson (AppenderFile), 6
- AppenderMemory, 15, 18, 28
- Appenders, 15, 17, 27, 34, 35, 43–46, 64
- AppenderTable, 5, 6, 8, 11, 12, 14, 17, 17, 28
- as.data.frame.event\_list (event\_list), 26
- as.data.frame.LogEvent, 18
- as.data.frame.LogEvent(), 44
- as.data.table.event\_list (event\_list), 26
- as.data.table.LogEvent (as.data.frame.LogEvent), 18
- as\_event\_list (event\_list), 26
- as\_LogEvent, 20, 27, 44, 65
- as\_logger\_config (logger\_config), 54
- as\_tibble.LogEvent (as.data.frame.LogEvent), 18
- base::as.data.frame(), 19, 27
- base::Filter(), 25
- base::format.POSIXct(), 37
- base::sprintf(), 46, 52
- base::sys.call(), 31
- basic\_config, 21
- CannotInitializeAbstractClassError, 22
- cli::tree(), 61
- colorize\_levels, 23, 35
- colorize\_levels(), 40
- condition, 48
- console\_threshold (simple\_logging), 62
- crayon::crayon, 23, 58
- data.table::data.table, 19
- data.tables, 18

- default\_exception\_handler, 24
- event\_list, 21, 26, 44, 65
- EventFilter, 24, 27, 28, 34
- EventFilters, 24, 28
- Filter (EventFilter), 24
- Filterable, 17, 18, 27, 34
- FilterForceLevel, 25, 29
- FilterInject, 25, 30
- Filters, 46
- format.ancestry (print.Logger), 59
- format.LogEvent, 21
- format.LogEvent (print.LogEvent), 57
- format.LogEvent(), 36, 37
- format.Logger (print.Logger), 59
- format.logger\_tree (print.logger\_tree), 60
- format.POSIXct(), 21, 42, 58
- get\_caller, 31
- get\_log\_levels, 33
- get\_log\_levels(), 35
- get\_logger, 32
- get\_logger(), 44, 49, 69
- get\_logger(name), 46
- get\_logger\_glue (get\_logger), 32
- get\_user (get\_caller), 31
- glue::glue, 39
- glue::glue(), 39, 52
- is\_filter, 34
- is\_filter(), 25, 28, 34
- is\_log\_level (standardize\_threshold), 64
- is\_log\_levels (standardize\_threshold), 64
- is\_threshold (standardize\_threshold), 64
- jsonlite::stream\_in(), 61
- jsonlite::toJSON(), 41, 42
- label\_levels, 23, 34
- Layout, 3, 5, 6, 17, 35, 39, 40, 42
- LayoutFormat, 5, 6, 8, 17, 36, 36, 39, 40, 42
- LayoutGlue, 36, 39, 39, 42
- LayoutJson, 6, 8, 36, 39, 40, 41, 61
- Layouts, 15, 17, 27, 43, 64
- Layouts (Layout), 35
- lgr::Appender, 3, 5–7, 9, 11, 13, 15, 17
- lgr::AppenderFile, 7, 9, 11, 13
- lgr::AppenderFileRotating, 11, 13
- lgr::AppenderMemory, 3
- lgr::EventFilter, 29, 30
- lgr::Filterable, 3, 5–7, 9, 11, 13, 15, 17, 45, 52
- lgr::Layout, 37, 39, 41
- lgr::Logger, 52
- log\_level, 21, 29, 49, 63
- log\_exception (simple\_logging), 62
- log\_level, 15, 43, 63
- log\_level (get\_log\_levels), 33
- log\_levels, 46
- log\_levels (get\_log\_levels), 33
- LogEvent, 15, 20, 21, 27–30, 36, 38, 39, 43, 44, 46, 58, 65, 67
- LogEvent\$new(), 46
- LogEvents, 15, 16, 26, 27, 30, 35, 49, 68
- LogEvents (LogEvent), 43
- Logger, 3, 4, 21, 24, 32, 43, 44, 54, 62, 63, 68, 69
- logger\_config, 48, 54
- logger\_index, 55
- logger\_index(), 56
- logger\_tree, 55, 61
- logger\_tree(), 55
- LoggerGlue, 52
- Loggers, 24, 27, 34, 64
- Loggers (Logger), 44
- pad\_left (pad\_right), 56
- pad\_left(), 40
- pad\_right, 56
- pad\_right(), 40
- POSIXct, 42, 43, 46
- print.ancestry (print.Logger), 59
- print.Appender, 57
- print.LogEvent, 57
- print.Logger, 59
- print.logger\_tree, 60
- R6::R6, 22, 24, 28, 34
- R6ClassGenerator, 32
- read\_json\_lines, 61
- read\_json\_lines(), 42
- remove\_appender (simple\_logging), 62
- remove\_log\_levels (get\_log\_levels), 33
- rotor::rotate(), 9–12, 14
- rotor::rotate\_date(), 12
- rotor::rotate\_time(), 14

- show\_data (simple\_logging), 62
- show\_dt (simple\_logging), 62
- show\_log (simple\_logging), 62
- show\_log(), 21
- simple\_logging, 62
- standardize\_log\_level
  - (standardize\_threshold), 64
- standardize\_log\_levels
  - (standardize\_threshold), 64
- standardize\_threshold, 21, 27, 44, 64
- string\_repr, 65
- suspend\_logging, 66
- system\_infos (get\_caller), 31
  
- threshold, 17, 18
- threshold (simple\_logging), 62
- tibble::tibble, 19
- tibbles, 18
- toString.LogEvent, 67
  
- unlabel\_levels (label\_levels), 34
- unsuspend\_logging (suspend\_logging), 66
- use\_logger, 68
  
- warnings, 48
- whoami::whoami(), 31
- with\_log\_level, 68
- with\_log\_level(), 25, 29
- with\_log\_value (with\_log\_level), 68
- with\_log\_value(), 25, 30
- with\_logging (suspend\_logging), 66
- without\_logging (suspend\_logging), 66